Sameer Wagh*, Divya Gupta, and Nishanth Chandran

SecureNN: 3-Party Secure Computation for Neural Network Training

Abstract: Neural Networks (NN) provide a powerful method for machine learning training and inference. To effectively train, it is desirable for multiple parties to combine their data – however, doing so conflicts with data privacy. In this work, we provide novel three-party secure computation protocols for various NN building blocks such as matrix multiplication, convolutions, Rectified Linear Units, Maxpool, normalization and so on. This enables us to construct three-party secure protocols for training and inference of several NN architectures such that no single party learns any information about the data. Experimentally, we implement our system over Amazon EC2 servers in different settings.
Our work advances the state-of-the-art of secure computation for neural networks in three ways:

1. Scalability: We are the first work to provide neural network training on Convolutional Neural Networks (CNNs) that have an accuracy of $> 99\%$ on the MNIST dataset;
2. Performance: For secure inference, our system outperforms prior 2 and 3-server works (SecureML, MiniONN, Chameleon, Gazelle) by 6×-113× (with larger gains obtained in more complex networks). Our total execution times are $2-4\times$ faster than even just the online times of these works. For secure training, compared to the only prior work (SecureML) that considered a much smaller fully connected network, our protocols are 79× and 7× faster than their 2 and 3-server protocols. In the WAN setting, these improvements are more dramatic and we obtain an improvement of 553×!
3. Security: Our protocols provide two kinds of security: full security (privacy and correctness) against one semi-honest corruption and the notion of privacy against one malicious corruption [Araki *et al.* CCS'16]. All prior works only provide semi-honest security and ours is the first system to provide any security against malicious adversaries for the secure computation of complex algorithms such as neural network inference and training.

Our gains come from a significant improvement in communication through the elimination of expensive garbled circuits and oblivious transfer protocols.

Keywords: Secure Multi-Party Computation, Privacy-preserving deep learning

DOI 10.2478/popets-2019-0035
Received 2018-11-30; revised 2019-03-15; accepted 2019-03-16.

1 Introduction

Neural networks (NN) have proven to be a very effective tool to produce predictive models that are widely used in applications such as healthcare, image classification, finance, and so on. The accuracy of these models get better as the amount of training data increases [41]. Large amounts of training data can be obtained by pooling in data from multiple contributors, but this data is sensitive and cannot be revealed in the clear due to proprietary reasons or compliance requirements [5, 15]. To enable training of NN models with good accuracy, it is highly desirable to securely train over data from multiple contributors such that plaintext data is kept hidden from the training entities.

In this work, we provide a solution for the above problem in the N server model. We model the problem as follows: a set of M data owners wish to execute training over their joint data using N servers. First, these M parties send "secret shares" of their input data to the N servers. The servers collectively run an interactive protocol to train a neural network over the joint data to produce a trained model that can be used for inference. The security requirement is that no individual party or server learns any information about any other party's training data. We call this the *N-server model*. We focus on the setting of $N = 3$, while M can be arbitrary and develop protocols specific for the $N = 3$ servers setting.

***Corresponding Author: Sameer Wagh:** Princeton University, E-mail: swagh@princeton.edu. Work done primarily at Microsoft Research, India
Divya Gupta: Microsoft Research, India, E-mail: divya.gupta@microsoft.com
Nishanth Chandran: Microsoft Research, India, E-mail: nichandr@microsoft.com

BY-NC-ND

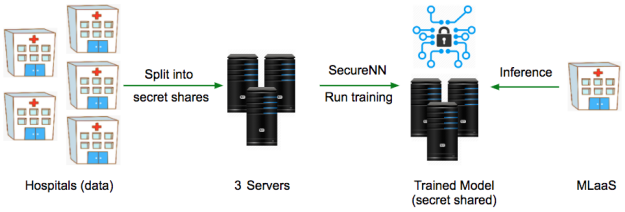

Fig. 1. Architecture

The trained model can be kept hidden from any single server/party and retained as secret shares between the servers (or reconstructed to obtain the model in the clear). Even if the model is retained as secret shares between the N servers, the inference/prediction can still be executed using the trained model on any new input – keeping the model, the new input, and the predicted output private from the other parties as well as the servers. For example, a group of M hospitals, each having sensitive patient data (such as heart rate readings, blood group, sugar levels etc.) can use the above architecture to train a model to run Machine Learning as a Service (MLaaS) and help predict some disease or irregular health behavior. The system can be set up such that the patient's sensitive input and predicted output are only revealed to the patient, and remains hidden from everyone else. The architecture is in Figure 1.

Secure multi-party computation (MPC) [8, 17, 25, 40] and specifically 3-party computation [6, 9, 10, 14, 18, 23, 28, 32] provide a generic solution for the above problem. However, general purpose MPC/3PC work over low-level circuits (either arithmetic or boolean) and for complex tasks such as neural network training, this leads to highly inefficient protocols that can take "forever to execute" (also pointed out by [21, 26, 33, 36]).

1.1 Our Contributions

In this work, we build specialized protocols in the 3-server setting that are tailored to popular functions in neural networks and improve the state-of-the-art in confidential machine learning in three ways. First, these protocols help us demonstrate for the first time the practicality of MPC to broad class of NN training algorithms including the rich class of Convolutional Neural Networks (CNN). Our techniques are powerful enough to train CNNs that produce an inference accuracy of greater than 99% on the MNIST dataset [3]. In contrast, SecureML [33], the prior state-of-the-art on secure NN training, considered only a much smaller fully connected network that gave an accuracy of 93.4% on MNIST dataset. Second, our protocols when evaluated on benchmarks considered in prior works such as [26, 30, 33, 36] outperforms by at least 6× and up to 533×. Third, while all prior works could only provide security against semi-honest adversaries, our protocols also give meaningful security against malicious adversaries, namely, *privacy against malicious adversaries* (formalized by [6]). Below, we elaborate on each of these points.

Scalability: Our main technical contribution is the construction of efficient 3-party protocols for various functions commonly used in machine learning algorithms – linear layer and convolutions (that are essentially matrix multiplications), Rectified Linear Unit (ReLU), Maxpool, normalization and their derivatives[1]. These protocols are compatible with each other and for a given NN training or inference algorithm, these can be *combined efficiently* to give the required secure computation protocol. The modularity of our protocols allows us to easily run experiments on a variety of neural networks including a 4-layer LeNet CNN [27].

Performance. Computing neural networks requires repeated computation of a mix of linear layers or convolutions followed by non-linear activation functions such as ReLU and Maxpool. All prior works [26, 30, 33] use a secure computation protocol for arithmetic circuits to compute the linear layers and Yao's garbled circuits to compute the non-linear activations. These two protocols are not very compatible with each and prior works use interconversion protocols to switch between arithmetic and garbled circuit encodings. Moreover, as noted by prior works [6, 33], garbled circuits incur a multiplicative overhead proportional to the security parameter in communication and are a major source of inefficiency. Our contribution is to construct new and efficient protocols for non-linear functions such as ReLU and Maxpool that completely avoid the use of garbled circuits and give at least 8× improvement in communication complexity. Furthermore, all our protocol maintain the invariant that parties start with shares of input over the ring $\mathbb{Z}_{2^{64}}$ (for system efficiency) and end with shares of output over the same ring. Hence, these protocols are inherently compatible with the protocol for linear layers and we get rid of all interconversion protocols as well. We discuss our concrete performance

1 We can also support other non-linear activation functions such as Leaky ReLU or piecewise linear functions.

improvements over prior works in Section 1.2.

Security. Our protocols provide two types of security. First, they provide full security (privacy and correctness) against the semi-honest corruption of a single server and any subset of clients, i.e., no server (together with any subset of clients) can learn any information about the inputs of the honest clients when it follows the protocol honestly[2]. Second, they provide privacy against any single malicious server, a notion formalized by Araki *et al.* [6]. Privacy against malicious server informally guarantees that a malicious server cannot learn anything about the inputs or outputs of the honest clients even if it deviates arbitrarily from the protocol specification (as long as the outputs of the computation are not revealed to the adversary).

All our protocols are fundamentally information-theoretically secure (i.e., adversaries can be computationally unbounded). Hence, for $N = 3$, as considered in our work, single corruption is the best corruption threshold that one could hope to achieve[3] [19]. However, in practice, we use pseudorandom functions to generate shared randomness as well as point-to-point secure channels between all pairs of parties thereby relying on computational assumptions for the implementation.

1.2 Experimental Results

To illustrate the generality and performance of our protocols, we consider the following 3 neural networks for the MNIST dataset – (A) a 3-layer DNN from SecureML [33], (B) a 4-layer CNN from MiniONN [30] and (C) a 4-layer LeNet network [27]. We evaluate our protocols both on secure training and secure prediction in the LAN/WAN settings and provide details below.

Secure Training. We train all the above networks on the MNIST dataset [3]. The *overall execution time* of our MPC protocol for Network A model over a LAN network is roughly an hour. For our largest CNN (Network C) our secure protocols execute in under 10 hours to achieve $> 98\%$ accuracy and in under 30 hours to achieve $> 99\%$ accuracy.

Comparison with prior work. We remark that this is the first work that implements training for networks B and C and these networks are much larger and give much higher accuracy ($> 98\%$) than network A (93%) considered by prior work [33]. The only prior work to consider secure training of neural networks is SecureML [33] that provides *computationally secure* protocols against a single semi-honest adversary in the 2-server and 3-server models for Network A. Compared to their 2-server protocols, we give an improvement of 79× and 553× in the LAN and WAN settings, respectively. They implement their 3-server protocol in the LAN setting only, and our protocols outperform this by 7×. SecureML also split their protocols into an offline (independent of data) and online phase. Even when comparing *their online time only with our total time*, we obtain a 2.8× improvement over their 3-server protocols. Our drastic improvements can be attributed to a roughly 8× improvement in communication complexity for computing non-linear functions and the elimination of expensive oblivious transfer protocols from the offline phase, which are another major source of overhead.

Secure Inference. Next, we consider the problem of secure inference for the same networks when the trained model is secret shared between the servers. For the smallest network A, a single prediction takes roughly $0.04s$ and $2.43s$ in the LAN and WAN settings, respectively. For the largest network C, a single prediction takes $0.23s$ in LAN and $4.08s$ in the WAN setting. As is observed by previous works as well, doing batch predictions is much faster in the amortized sense than doing multiple predictions serially. For instance, for network C, a batch of 128 predictions take only $10.82s$ in the LAN and $30.45s$ in the WAN setting.

Comparison with prior work. There has been a large effort on this problem, both in the 2-server [26, 30, 33] and 3-server settings [36]. As our experiments show, the total execution time of our protocols are 113× faster than [33], 71.7× faster than MiniONN [30], 35.5× faster than Chameleon [36] and 6.23× faster than Gazelle [26]. Our *total execution times are also faster than just the online execution times* of these protocols and we obtain improvements of 4.2× over SecureML, 44.15× over MiniONN, 17.9× over Chameleon and 2.5× over Gazelle. These gains come from a $74.2\times, 3.2\times$ and 7.9× reduction in communication over MiniONN, Chameleon, and Gazelle, respectively[4].

2 When the trained model is revealed to the adversary, we give the standard guarantee that nothing is revealed about honest clients' inputs beyond the model.

3 Secure computation protocols that tolerate a collusion of two servers (i.e., dishonest majority setting) require heavy public key cryptographic tools such as oblivious transfer.

4 [33] do not explicitly list their communication complexity.

Concurrent and Independent Work. In concurrent and independent work, ABY^3 [31] achieve similar results but using techniques fundamentally different from ours. They theoretically describe how to convert their protocols to achieve malicious security and provide implementation and experimental numbers for semi-honest secure protocols. In contrast, our performance numbers are for both semi-honest security as well as malicious privacy. SecureNN protocols are extremely simple to implement giving them an advantage in real world deployment (as they do not require the heavy optimizations that are required when using garbled circuits). A more detailed comparison is provided in Section 7.

1.3 Organization of the paper

We begin with a high-level technical overview of our protocols in Section 2. Next, we describe the security model and the neural network training algorithms that we use in Section 3. Section 4 contains our low-level 3-server protocols that are used as building blocks in our main protocols for various functionalities. In Section 5, we describe protocols for all machine learning functions such as matrix multiplication, convolution, ReLU (its derivative), Maxpool (its derivative) and so on. We discuss theoretical efficiency of our protocols in Section 6. Finally, we present a detailed evaluation of our experiments in Section 7. We discuss related works in Appendix A, provide details on the number encoding and network architectures in Appendix B, C, and provide security proofs in Appendix D.

2 Technical Overview

Secure protocols for neural network algorithms generally follow the paradigm of executing arithmetic computation, such as matrix multiplication and convolutions, using Beaver triplets or homomorphic encryption and executing Boolean computation, such as ReLU, Maxpool and its derivatives, using Yao's garbled circuits. In order to make these protocols compatible with each other, share conversion protocols are also used to move from an arithmetic encoding (either arithmetic sharing or homomorphic encryption ciphertext) to a Boolean encoding (garbled encoding) and vice-versa. The communication cost of securely evaluating Boolean computations is prohibitive due to the use of Yao's garbled circuits that incur a multiplicative factor overhead of 128 (the security parameter, κ). This is precisely where our new protocols come to the rescue. We develop new protocols for Boolean computation (such as ReLU, Maxpool and its derivatives) that have much lesser communication overhead (at least 8× better) than the cost of converting to a Yao encoding and executing a garbled circuit. We now present our techniques in more detail.

We denote the three servers by P_0, P_1 and P_2. At the start of any protocol, parties P_0 and P_1 hold 2-out-of-2 additive secret shares of the inputs to the protocol. All our protocols maintain the invariant that at the end of the protocol P_0 and P_1 hold 2-out-of-2 shares of the output. We stress that even though for all our protocols only P_0 and P_1 hold the shares of the input and the output, P_2 also crucially takes part in the real computation during the protocol. That is, P_2 is not a dummy party that only assists in the two-party protocol between P_0 and P_1 by providing relevant randomness.

Non-linear activations. We first describe our main ideas for computing the derivative of ReLU function, that is ReLU'.

Function ReLU'. Note that $\mathsf{ReLU}'(x)$ is 1 if $x \geq 0$ and 0 otherwise. First, we note that $\mathsf{ReLU}'(x)$ is closely related to the most-significant bit (MSB)[5] of x in our representation of values in $\mathbb{Z}_{2^{64}}$. That is, $\mathsf{ReLU}'(x)$ is 1 iff $\mathsf{MSB}(x) = 0$. Hence, it suffices to compute the $\mathsf{MSB}(x)$. Next, since computing LSB of a number is much easier than computing the MSB (as it does not require bit extraction), we flip the problem to computing LSB as follows: $\mathsf{MSB}(a) = \mathsf{LSB}(2a)$ if we are working over an odd ring[6]. For now, let us assume that we are working over an odd ring and we later describe how we go from even ring $\mathbb{Z}_{2^{64}}$ to an odd ring $\mathbb{Z}_{2^{64}-1}$.

At the start of the protocol, P_0 and P_1 hold shares of a (over $\mathbb{Z}_{2^{64}-1}$), using which they locally compute shares of $y = 2a$. Now, P_2 would assist in computing the LSB of y as follows: From now on, we denote $\mathsf{LSB}(y) = y[0]$. The first observation is that for three numbers u, v, w such that $u = v + w$, $u[0] = v[0] \oplus w[0]$ if the addition does not "wrap around" the ring and $u[0] = u[0] \oplus v[0] \oplus 1$ if addition wraps around. The second observation is that if x is a random number chosen by P_2 and is unknown to P_0 and P_1, then it is okay for P_0, P_1 to learn $r = y + x$.

5 $\mathsf{MSB}(x)$ is the leftmost bit in the 64-bit representation of x.

6 In a group of order n, we have $\mathsf{MSB}(x) = 1$ iff $x > n/2$ iff $n > 2x - n > 0$; if n is odd, then so is $2x - n$ and it follows that $\mathsf{MSB}(x) = 1$ iff $\mathsf{LSB}(2x) = 1$.

This is because the secret y is masked by random x. Hence, P_2 gives secret shares of x as well as shares of $x[0]$ to P_0, P_1 and they reconstruct r. Now, all that is left is to figure out whether the addition $y + x$ wraps around the ring or not. For this, the third observation is that this addition wraps around if and only if the final sum is less than one of the individual values – that is, it wraps around iff $x > r$. Thus, if we can compute shares of $x > r$ between P_0 and P_1, then we are done.

Next, we construct a protocol for comparison. We first define a functionality called *private compare* (denoted by $\mathcal{F}_{\text{PC}}$). This three-party functionality assumes that P_0 and P_1 each have a share of the bits of $\ell-$bit value x (over field $\mathbb{Z}_p$) as well as a common number r and a random bit β as input. It computes the bit $(x > r)$ (which is 1 if $x > r$ and 0 otherwise) and XOR masks it with the bit β. This output $\beta \oplus (x > r)$ is given to P_2. We implement this functionality by building on the techniques of [20, 35] and provide a more efficient variant. Note that this protocol requires parties to have shares of bits of x over field $\mathbb{Z}_p$. These are provided to P_0, P_1 by P_2. With these protocols, we are ready to compute the $\mathsf{ReLU}'(\cdot)$ function if P_0 and P_1 began with shares of the input over an odd ring.

Now, we revisit the requirement of an odd ring. As we explained above, all of this works, if we had shares of a over an odd ring. Now, we could execute our protocol over the ring $\mathbb{Z}_N$ with N being odd. However doing so is fairly inefficient as matrix multiplication over the ring $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) is much faster. This is because (as observed in [33]), native implementation of matrix multiplication over `long` (or `int`) automatically implements the modulo operation over $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) and many libraries heavily optimize matrix multiplication over these rings, which give significant efficiency improvements compared to operations over any other ring. Hence, we provide a protocol that converts values ($\neq L-1$) that are secret shared over $\mathbb{Z}_L$ into shares over $\mathbb{Z}_{L-1}$. This protocol also uses the private compare protocol and may be of independent interest.

Finally, this design (and our protocol) enables us to run our comparison protocol (the protocol that realizes $\mathcal{F}_{\text{PC}}$ above) over a *small* field $\mathbb{Z}_p$ (we choose $p = 67$ concretely) and this reduces the communication complexity significantly. Using these protocols, we obtain our protocol for computing $\mathsf{ReLU}'(x)$ (the derivative of ReLU) beginning with shares over $\mathbb{Z}_{2^{64}}$.

Figure 2 shows the different secret sharing schemes used in protocols in SecureNN. Specifically, it shows how various secret sharing schemes are used in ReLU and Private Compare protocols.

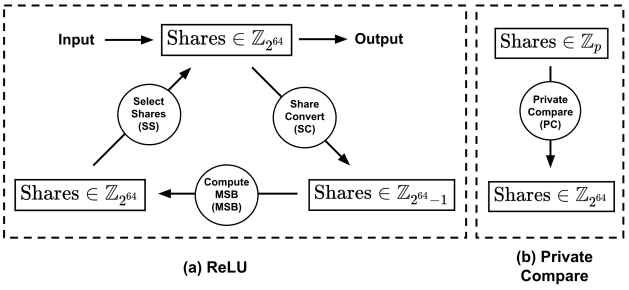

Fig. 2. Flow of different types of secret sharing schemes used in (a) ReLU (b) Private Compare.

Other non-linear functions. In this work, we describe protocols for ReLU, Maxpool, its derivative and normalization or division. Maxpool, ReLU and division are implemented using ReLU' and multiplication. Similar ideas can be used to obtain efficient protocols for other non-linear activation functions such as PReLU, LeakyReLU, piecewise linear functions (used to approximate sigmoid) and their derivatives. We also construct an efficient protocol for the derivative of Maxpool exploiting specific number-theoretic properties.

Matrix multiplication and Convolutions. An information-theoretic matrix multiplication protocol over shares when 3 parties are involved is straightforward using matrix-based Beaver multiplication triplets [7] and is omitted from the discussion here. For our implementation we use Beaver triplets generated using PRFs. Convolutions are implemented in a very similar manner to matrix multiplication.

Privacy against malicious adversaries. Since our protocols are fundamentally information-theoretic, it is easy to show that all messages exchanged in the protocol are individually uniformly random. As noted by Araki *et al.* [6], this property then suffices to show that any two executions of the protocol with different inputs are indistinguishable to the malicious adversary and subsequently that the same protocols provide privacy against malicious adversaries. This guarantees that a malicious server cannot learn anything about clients' inputs even if it deviates arbitrarily from the protocol.

3 Preliminaries

3.1 Threat Model and Security

In this work, we consider full semi-honest security as well as privacy against malicious adversaries.

Semi-honest Security. A *semi-honest* (also known as honest-but-curious) adversary follows the protocol specifications honestly but tries to learn information from the protocol. We consider a semi-honest adversary that corrupts a single server (and any number of clients) and prove full security (i.e., privacy and correctness) of our protocols in the simulation paradigm [12, 13, 25]. The universal composability framework [12] allows one to guarantee the security of arbitrary composition of different protocols. Hence, we can prove the security of individual protocols and the security of end-to-end neural network algorithms follows from the composition. Security is modeled by defining two interactions: a *real* interaction where the parties execute a protocol in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$, and an *ideal* interaction where parties send their inputs to a trusted functionality $\mathcal{F}$ that conducts the desired computation truthfully. Security requires that for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ (called the simulator) in the ideal interaction, such that no environment $\mathcal{Z}$ can distinguish between real and ideal interactions. A protocol Π is said to *securely realize* a functionality $\mathcal{F}$ if for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ in the ideal interaction, such that no environment $\mathcal{Z}$, on any input, can tell apart the real interaction from the ideal interaction, except with negligible probability (in the security parameter κ).

Privacy against Malicious Adversary. A *malicious* adversary can arbitrarily deviate from the protocol specification. Araki *et al.* [6] formalized the notion of privacy against malicious adversaries in the client-server model. In this model, similar to our setting, the clients secret share their inputs between the servers, the servers run the secure computation to compute the shares of the output. Hence, the servers running the secure computation do not get to see the input or the output. Intuitively, privacy against malicious server guarantees that even a malicious adversary cannot break the privacy of inputs or outputs of the honest parties. This is formalized using an indistinguishability-based argument that says that for any two inputs of the honest parties, the view of the adversary in the protocol is indistinguishable. For our protocols, we prove that they satisfy privacy against a malicious adversary that corrupts any one of the three servers. Even though this notion is weaker than full simulation based malicious security (in particular, this does not guarantee correctness in the presence of malicious behavior), it does guarantee that privacy is not compromised by malicious behavior.

3.2 Notation

In our protocols, we use additive secret sharing over the three rings $\mathbb{Z}_L$, $\mathbb{Z}_{L-1}$, and $\mathbb{Z}_p$, where $L = 2^\ell$ and p is a prime. Note that $\mathbb{Z}_{L-1}$ is a ring of odd size and $\mathbb{Z}_p$ is a field. Specifically, we use following the three types of secret sharings in the following settings:

(A) **Additive shares in $\mathbb{Z}_L$:** Additive shares of a number in ring $\mathbb{Z}_L$. In this work $L = 2^{64}$.

(B) **Additive shares in $\mathbb{Z}_{L-1}$:** Additive shares of a number in ring $\mathbb{Z}_{L-1}$.

(C) **Additive shares of bits in $\mathbb{Z}_p$:** Each bit of the 64-bit secret is additively shared in $\mathbb{Z}_p$. In this work $p = 67$. In other words, each 64-bit number is shared as a vector of 64 shares, with each share being a value between 0 and 66 (inclusive).

We use 2-out-of-2 secret sharing and use $\langle x \rangle_0^t$ and $\langle x \rangle_1^t$ to denote the two shares of x over $\mathbb{Z}_t$ – specifically, the scheme generates $r \xleftarrow{\$} \mathbb{Z}_t$, sets $\langle x \rangle_0^t = r$ and $\langle x \rangle_1^t = x - r \pmod t$. We use $\langle x \rangle^t$ to denote sharing of x over $\mathbb{Z}_t$. The algorithm $\mathsf{Share}^t(x)$ generates the two shares of x over $\mathbb{Z}_t$ and algorithm $\mathsf{Reconst}^t(x_0, x_1)$ reconstructs a value x using x_0 and x_1 as the two shares (reconstruction is simply $x_0 + x_1$ over $\mathbb{Z}_t$). For an $\ell-$bit integer x, we use $x[i]$ to denote the i^{th} bit of x and $\{\langle x[i] \rangle^t\}_{i \in [\ell]}$ to denote the shares of bits of x over $\mathbb{Z}_t$. For an $m \times n$ matrix X, $\langle X \rangle_0^t$ and $\langle X \rangle_1^t$ denote the matrices that are created by secret sharing the elements of X component-wise (other matrix notation such as $\mathsf{Reconst}^t(X_0, X_1)$ is similarly defined component-wise).

3.3 Neural Networks

Our main focus in this work is on Deep and Convolutional Neural Network (DNN and CNN) training algorithms. At a very high level, every layer in the forward propagation comprises of a linear operation (such as matrix multiplication in the case of fully connected layers and convolution in the case of Convolutional Neural Networks, where weights are multiplied by the activation), followed by a (non-linear) activation function f. One of the most popular activation functions is the Rectified Linear Unit (ReLU) defined as $\mathsf{ReLU}(x) = \mathsf{max}(0, x)$. The backpropogation updates the weights appropriately making use of derivative of the activation function (in this case $\mathsf{ReLU}'(x)$, which is defined

to be 1 if $x > 0$ and 0 otherwise) and matrix multiplication. Cross entropy is used as the loss function and stochastic gradient descent minimizes the loss.

A large class of networks can be represented using the following functions: matrix multiplication, convolution, ReLU, MaxPool (defined as the maximum of a set of values, usually in a sub-matrix), normalization (defined as $\frac{x_i}{\sum x_i}$ for a given set of values $\{x_1, \cdots, x_n\}$) and their derivatives. In this work, we consider three neural networks for training - (A) a 3 layer DNN (same as the neural network in [33]) that provides an inference accuracy of 93.4% on the MNIST dataset [3] (after training for 15 epochs) (B) a 4-layer network from MiniONN [30] and (C) a 4-layer LeNeT network that provides an inference accuracy of $> 99\%$ on the same data set (after training for 15 epochs). For inference, we additionally consider a neural network from Chameleon [36]. More details on these networks are presented in Section 7.3.

3.4 Protocols Structure

First, in Section 4, we provide protocols for "supporting functionalities" that will be used as building blocks to construct protocols for our main functionalities. In Section 5, we provide protocols for realizing our "main functionalities" that correspond to various neural network layers such as linear layer, convolution layer, ReLU, and so on. These protocols along with their dependencies are presented in Figure 3. In Section 5.7, we outline how to put these main protocols together to obtain protocols for a large class of neural networks. We also provide overview of proofs of security and refer to appendix for full proofs. In Appendix E, we argue that all our protocols satisfy the privacy against a malicious corruption of a single server in the client-server model as defined by [6].

We assume that any pair of parties pre-share fresh shares of 0. This is trivial to do – the two parties exchange a PRF key, k, and one party sets its share to $z_0 = \mathsf{PRF}_k(\mathsf{ctr})$, while the other sets it share to $z_1 = -\mathsf{PRF}_k(\mathsf{ctr})$, where ctr is a known counter value. When we use the term "fresh share" of some value x, we mean that the randomness used to generate the share of x has not been used anywhere else in that or any other protocol. We say "party P_i generates shares $\langle x \rangle_j^t$ for $j \in \{0,1\}$ and sends to P_j" to mean "party P_i generates $(\langle x \rangle_0^t, \langle x \rangle_1^t) \leftarrow \mathsf{Share}^t(x)$ and sends $\langle x \rangle_j^t$ to P_j for $j \in \{0,1\}$".

In all our main protocols (Section 5), we maintain the invariant that parties P_0 and P_1 begin with "fresh" shares of input value (over $\mathbb{Z}_L$) and output a "fresh" share of the output value (again over $\mathbb{Z}_L$) at the end of the protocol – this will enable us (as shown in Section 5.7) to arbitrarily compose our main protocols to obtain protocols for a variety of neural networks. Party P_2 takes the role of "assistant" in all protocols and has no input to protocols. In the supporting protocols alone, P_2 sometimes receives an output.

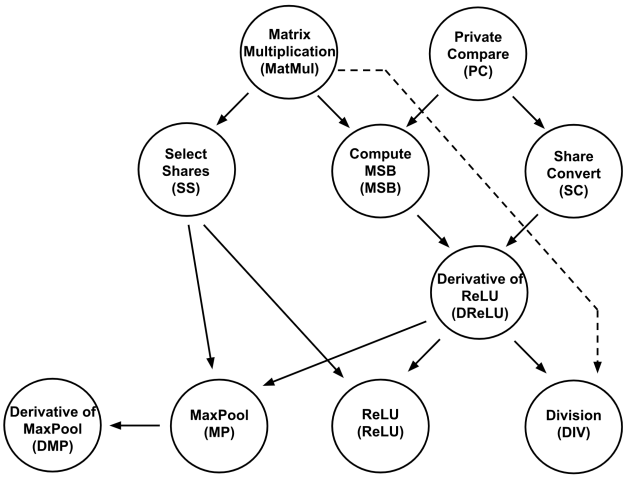

Fig. 3. Functionality dependence of protocols in SecureNN

4 Supporting Protocols

In this section, we describe various building blocks to our main protocols. Some of these protocols deviate from the invariant described above – i.e., P_0 and P_1 do not necessarily begin/end protocols with shares of input/output over $\mathbb{Z}_L$. Further, P_2 receives output in these protocols. Due to lack of space, the formal description of the functionalities realized by these protocols is given in the full version [38]. We provide the proofs of security for our protocols in Appendix D, E, and F. We start with the simplest protocols (such as those for matrix multiplication) and gradually build other protocols that are used in the computation of non-linear functions.

4.1 Matrix Multiplication

Algorithm 1 describes our protocol for secure multiplication (functionality $\mathcal{F}_{\text{MATMUL}}$) where parties P_0 and P_1 hold shares of $X \in \mathbb{Z}_L^{m \times n}$ and $Y \in \mathbb{Z}_L^{n \times v}$ and the functionality outputs fresh shares of $Z = X \cdot Y$ to P_0, P_1.

Intuition. Our protocol relies on standard cryptographic technique for multiplication of using Beaver

Algorithm 1 Mat. Mul. $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$:

Input: P_0 & P_1 hold $(\langle X\rangle_0^L, \langle Y\rangle_0^L)$ & $(\langle X\rangle_1^L, \langle Y\rangle_1^L)$ resp.

Output: P_0 gets $\langle X \cdot Y\rangle_0^L$ and P_1 gets $\langle X \cdot Y\rangle_1^L$.

Common Randomness: P_0 and P_1 hold shares of zero matrices over $\mathbb{Z}_L^{m\times v}$ resp.; i.e., P_0 holds $\langle 0^{m\times v}\rangle_0^L = U_0$ & P_1 holds $\langle 0^{m\times v}\rangle_1^L = U_1$

1: P_2 picks random matrices $A \xleftarrow{\$} \mathbb{Z}_L^{m\times n}$ and $B \xleftarrow{\$} \mathbb{Z}_L^{n\times v}$ and generates for $j \in \{0,1\}$, $\langle A\rangle_j^L, \langle B\rangle_j^L, \langle C\rangle_j^L$ and sends to P_j, where $C = A \cdot B$.

2: For $j \in \{0,1\}$, P_j computes $\langle E\rangle_j^L = \langle X\rangle_j^L - \langle A\rangle_j^L$ and $\langle F\rangle_j^L = \langle Y\rangle_j^L - \langle B\rangle_j^L$.

3: P_0 & P_1 reconstruct E & F by exchanging shares.

4: For $j \in \{0,1\}$, P_j outputs $-jE \cdot F + \langle X\rangle_j^L \cdot F + E \cdot \langle Y\rangle_j^L + \langle C\rangle_j^L + U_j$.

triplets [7] generalized to the matrix setting. P_2 generates these triplet shares and sends to parties P_0, P_1.

4.2 Select Share

Algorithm 2 describes our 3-party protocol realizing the select share functionality $\mathcal{F}_{\text{SS}}$, which is as follows: Parties P_0, P_1 hold shares of x, y over $\mathbb{Z}_L$. They also hold shares of a selection bit $\alpha \in \{0,1\}$ over $\mathbb{Z}_L$ ($L = 2^{64}$). Parties P_0, P_1 get fresh shares of x if $\alpha = 0$ and fresh shares of y if $\alpha = 1$ from the functionality.

Algorithm 2 SelectShare $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $(\langle\alpha\rangle_0^L, \langle x\rangle_0^L, \langle y\rangle_0^L)$ and $(\langle\alpha\rangle_1^L, \langle x\rangle_1^L, \langle y\rangle_1^L)$, resp.

Output: P_0, P_1 get $\langle z\rangle_0^L$ and $\langle z\rangle_1^L$, resp., where $z = (1-\alpha)x + \alpha y$.

Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1.

1: For $j \in \{0,1\}$, P_j compute $\langle w\rangle_j^L = \langle y\rangle_j^L - \langle x\rangle_j^L$

2: P_0, P_1, P_2 invoke $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle\alpha\rangle_j^L, \langle w\rangle_j^L)$ and P_0, P_1 learn $\langle c\rangle_0^L$ and $\langle c\rangle_1^L$, resp.

3: For $j \in \{0,1\}$, P_j outputs $\langle z\rangle_j^L = \langle x\rangle_j^L + \langle c\rangle_j^L + u_j$.

Intuition. We note that selecting between x and y can be arithmetically expressed as $(1-\alpha)\cdot x + \alpha \cdot y = x + \alpha \cdot (y-x)$. We compute the latter expression in our protocol using one call to Π_{MatMul} for multiplying α and $(y-x)$.

4.3 Private Compare

Algorithm 3 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{PC}}$ for comparison that is as follows: The parties P_0 and P_1 holds shares of bits of ℓ-bit integer x in $\mathbb{Z}_p$ ($p = 67$ and hence $\mathbb{Z}_p$ is a Field), i.e., $\{\langle x[i]\rangle_0^p\}_{i\in[\ell]}$ and $\{\langle x[i]\rangle_1^p\}_{i\in[\ell]}$, respectively. P_0, P_1 also hold an ℓ-bit integer r and a bit β. At the end of the protocol, P_2 learns a bit $\beta' = \beta \oplus (x > r)$, where $(x > r)$ denotes the bit which is 1 when $x > r$ over the integers and 0 otherwise.

Algorithm 3 PrivateCompare $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x[i]\rangle_0^p\}_{i\in[\ell]}$ and $\{\langle x[i]\rangle_1^p\}_{i\in[\ell]}$, respectively, a common input r (an ℓ bit integer) and a common random bit β.

Output: P_2 gets a bit $\beta \oplus (x > r)$.

Common Randomness: P_0, P_1 hold ℓ common random values $s_i \in \mathbb{Z}_p^*$ for all $i \in [\ell]$ and a random permutation π for ℓ elements. P_0 and P_1 additionally hold ℓ common random values $u_i \in \mathbb{Z}_p^*$.

1: Let $t = r + 1 \mod 2^\ell$.

2: For each $j \in \{0,1\}$, P_j executes Steps 3–14:

3: **for** $i = \{\ell, \ell-1, \ldots, 1\}$ **do**

4: **if** $\beta = 0$ **then**

5: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jr[i] - 2r[i]\langle x[i]\rangle_j^p$

6: $\langle c_i\rangle_j^p = jr[i] - \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{\ell} \langle w_k\rangle_j^p$

7: **else if** $\beta = 1$ AND $r \neq 2^\ell - 1$ **then**

8: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jt[i] - 2t[i]\langle x[i]\rangle_j^p$

9: $\langle c_i\rangle_j^p = -jt[i] + \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{\ell} \langle w_k\rangle_j^p$

10: **else**

11: If $i \neq 1$, $\langle c_i\rangle_j^p = (1-j)(u_i+1) - ju_i$, else $\langle c_i\rangle_j^p = (-1)^j \cdot u_i$.

12: **end if**

13: **end for**

14: Send $\{\langle d_i\rangle_j^p\}_i = \pi\left(\left\{s_i\langle c_i\rangle_j^p\right\}_i\right)$ to P_2

15: For all $i \in [\ell]$, P_2 computes $d_i = \mathsf{Reconst}^p(\langle d_i\rangle_0^p, \langle d_i\rangle_1^p)$ and sets $\beta' = 1$ iff $\exists i \in [\ell]$ such that $d_i = 0$.

16: P_2 outputs β'.

Intuition. Our starting point is the 2-party comparison present in [20, 35]. We build on this to give a much more efficient 3-party protocol. We want to compute $\beta' = \beta \oplus (x > r)$. That is, for $\beta = 0$, we compute $x > r$

and for $\beta = 1$, we compute $1 \oplus (x > r) \equiv (x \leq r) \equiv (x < (r+1))$ over integers. We discuss the corner case of $r = 2^\ell - 1$ below. In this case, $x \leq r$ is always true.

Consider the case of $\beta = 0$. In this case, $\beta' = 1$ iff $(x > r)$ or at the leftmost bit where $x[i] \neq r[i]$, $x[i] = 1$. We compute $w_i = x[i] \oplus r[i] = x[i] + r[i] - 2x[i]r[i]$ and $c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{\ell} w_k$. Since r is known to both P_0, P_1, shares of both w_i and c_i can be computed locally. Now, we can prove that $\exists\ i.c_i = 0$ iff $x > r$. Hence, both P_0, P_1 send shares of c_i to P_2 who reconstructs c_i and looks for a 0. To ensure security against a corrupt P_2, we hide exact values of non-zero c_i's and position of (a possible) 0 by multiplying with random s_i and permuting these values by a common permutation π. These s_i and π are common to both P_0 and P_1.

In the case when $\beta = 1$ and $r \neq 2^\ell - 1$, we compute $(r+1) > x$ using similar logic as above. In the corner case of $r = 2^\ell - 1$, both parties P_0, P_1 know that result of $x \leq r \equiv (r+1) > x$ over integers should be true. Hence, they together pick shares of c_i such that there is exactly one 0. This is done by P_0, P_1 having common values u_i that they use to create a valid share of a 0 and $\ell - 1$ shares of 1 (see Step 11). Note that for the re-randomization using s_i's to work, it is crucial that we work over a field such as $\mathbb{Z}_p$.

4.4 Share Convert

Algorithm 4 describes our three-party protocol for converting shares over $\mathbb{Z}_L$ to $\mathbb{Z}_{L-1}$ realizing the functionality $\mathcal{F}_{\text{SC}}$ (again, $L = 2^{64}$). Here, parties P_0, P_1 hold shares of $\langle a \rangle^L$ such that $a \neq L-1$. At the end of the protocol, P_0, P_1 hold fresh shares of same value over $L-1$, i.e., $\langle a \rangle^{L-1}$. In this algorithm, $\kappa := \mathsf{wrap}(x, y, L)$ is 1 if $x + y \geq L$ over integers and 0 otherwise. That is, κ denotes the wrap-around bit for the computation $x + y \mod L$.

Intuition: Let $\theta = \mathsf{wrap}(\langle a \rangle_0^L, \langle a \rangle_1^L, L)$. Now, we note that if $\theta = 1$, i.e., if the original shares wrapped around L, then we need to subtract 1, else original shares are also valid shares of same value of $L-1$. Hence, in the protocol we compute shares of bit θ over $L-1$ and subtract from original shares locally. This protocol makes use of novel modular arithmetic to securely compute these shares of θ, an idea which is potentially of independent interest. We explain these relations in the correctness proof below.

Lemma 1. *Protocol $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ in Algorithm 4 securely realizes $\mathcal{F}_{\text{SC}}$.*

Proof. For correctness we need to prove that $\mathsf{Reconst}^{L-1}(\langle y \rangle_0^{L-1}, \langle y \rangle_1^{L-1}) = \mathsf{Reconst}^{L}(\langle a \rangle_0^{L}, \langle a \rangle_1^{L}) = a$. Looking at Step 11 of the protocol and the intuition above, it suffices to prove that $\mathsf{Reconst}^{L-1}(\langle \theta \rangle_0^{L-1}, \langle \theta \rangle_1^{L-1}) = \theta = \mathsf{wrap}(\langle a \rangle_0^{L}, \langle a \rangle_1^{L}, L)$. First, by correctness of protocol Π_{PC}, $\eta' = \eta'' \oplus (x > r-1)$. Next, let $\eta = \mathsf{Reconst}^{L-1}(\langle \eta \rangle_0^{L-1}, \langle \eta \rangle_1^{L-1}) = \eta' \oplus \eta'' = (x > r-1)$. Next, note that $x \equiv a + r \mod L$. Hence, $\mathsf{wrap}(a, r, L) = 0$ iff $x > r-1$. By the correctness of wrap, following relations hold over the integers:

1. $r = \langle r \rangle_0^L + \langle r \rangle_1^L - \alpha L$.
2. $\langle \tilde{a} \rangle_j^L = \langle a \rangle_j^L + \langle r \rangle_j^L - \beta_j L$.
3. $x = \langle \tilde{a} \rangle_0^L + \langle \tilde{a} \rangle_1^L - \delta L$.
4. $x = a + r - (1 - \eta)L$.
5. Let θ be such that $a = \langle a \rangle_0^L + \langle a \rangle_1^L - \theta L$.

Computing, (1) − (2) − (3) + (4) + (5) gives us $\theta = \beta_0 + \beta_1 - \alpha + \delta + \eta - 1$. This is exactly, what the parties P_0 and P_1 calculate in Step 10. We prove security in Appendix F. □

Algorithm 4 ShareConvert $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$, respectively such that $\mathsf{Reconst}^L(\langle a \rangle_0^L, \langle a \rangle_1^L) \neq L-1$.

Output: P_0, P_1 get $\langle a \rangle_0^{L-1}$ and $\langle a \rangle_1^{L-1}$.

Common Randomness: P_0, P_1 hold a random bit η'', a random $r \in \mathbb{Z}_L$, shares $\langle r \rangle_0^L, \langle r \rangle_1^L, \alpha = \mathsf{wrap}(\langle r \rangle_0^L, \langle r \rangle_1^L, L)$ and shares of 0 over $\mathbb{Z}_{L-1}$ denoted by u_0 and u_1.

1: For each $j \in \{0, 1\}$, P_j executes Steps 2–3
2: $\langle \tilde{a} \rangle_j^L = \langle a \rangle_j^L + \langle r \rangle_j^L$ and $\beta_j = \mathsf{wrap}(\langle a \rangle_j^L, \langle r \rangle_j^L, L)$.
3: Send $\langle \tilde{a} \rangle_j^L$ to P_2.
4: P_2 computes $x = \mathsf{Reconst}^L(\langle \tilde{a} \rangle_0^L, \langle \tilde{a} \rangle_1^L)$ and $\delta = \mathsf{wrap}(\langle \tilde{a} \rangle_0^L, \langle \tilde{a} \rangle_1^L, L)$.
5: P_2 generates shares $\{\langle x[i] \rangle_j^p\}_{i \in [\ell]}$ and $\langle \delta \rangle_j^{L-1}$ for $j \in \{0, 1\}$ and sends to P_j.
6: P_0, P_1, P_2 invoke$^7\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $\left(\{\langle x[i] \rangle_j^p\}_{i \in [\ell]}, r-1, \eta''\right)$ and P_2 learns η'.
7: For $j \in \{0, 1\}$, P_2 generates $\langle \eta' \rangle_j^{L-1}$ and sends to P_j.
8: For each $j \in \{0, 1\}$, P_j executes Steps 9–11
9: $\langle \eta \rangle_j^{L-1} = \langle \eta' \rangle_j^{L-1} + (1-j)\eta'' - 2\eta'' \langle \eta' \rangle_j^{L-1}$
10: $\langle \theta \rangle_j^{L-1} = \beta_j + (1-j) \cdot (-\alpha - 1) + \langle \delta \rangle_j^{L-1} + \langle \eta \rangle_j^{L-1}$
11: Output $\langle y \rangle_j^{L-1} = \langle a \rangle_j^L - \langle \theta \rangle_j^{L-1} + u_j$ (over $L-1$)

4.5 Compute MSB

Algorithm 5 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{MSB}}$ that computes the most significant bit[8] (MSB) of a value $a \in \mathbb{Z}_{L-1}$. P_0, P_1 hold shares of a over odd ring $\mathbb{Z}_{L-1}$ and end with shares of $\mathsf{MSB}(a)$ over $\mathbb{Z}_L$.

Algorithm 5 ComputeMSB $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^{L-1}$ and $\langle a \rangle_1^{L-1}$, respectively.
Output: P_0, P_1 get $\langle \mathsf{MSB}(a) \rangle_0^L$ and $\langle \mathsf{MSB}(a) \rangle_1^L$.
Common Randomness: P_0, P_1 hold a random bit β and random shares of 0 over L, denoted by u_0 and u_1 resp.

1: P_2 picks $x \xleftarrow{\$} \mathbb{Z}_{L-1}$. Next, P_2 generates $\langle x \rangle_j^{L-1}$, $\{\langle x[i] \rangle_j^p\}_i$, $\langle x[0] \rangle_j^L$ for $j \in \{0,1\}$ and sends to P_j.
2: For $j \in \{0,1\}$, P_j computes $\langle y \rangle_j^{L-1} = 2\langle a \rangle_j^{L-1}$ and $\langle r \rangle_j^{L-1} = \langle y \rangle_j^{L-1} + \langle x \rangle_j^{L-1}$.
3: P_0, P_1 reconstruct r by exchanging shares.
4: P_0, P_1, P_2 call $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\left(\{\langle x[i] \rangle_j^p\}_{i \in [\ell]}, r, \beta\right)$ and P_2 learns β'.
5: P_2 generates $\langle \beta' \rangle_j^L$ and sends to P_j for $j \in \{0,1\}$.
6: For $j \in \{0,1\}$, P_j executes Steps 7–8
7: $\langle \gamma \rangle_j^L = \langle \beta' \rangle_j^L + j\beta - 2\beta \langle \beta' \rangle_j^L$
8: $\langle \delta \rangle_j^L = \langle x[0] \rangle_j^L + jr[0] - 2r[0] \langle x[0] \rangle_j^L$
9: P_0, P_1, P_2 call $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \gamma \rangle_j^L, \langle \delta \rangle_j^L)$ and P_j learns $\langle \theta \rangle_j^L$.
10: For $j \in \{0,1\}$, P_j outputs $\langle \alpha \rangle_j^L = \langle \gamma \rangle_j^L + \langle \delta \rangle_j^L - 2\langle \theta \rangle_j^L + u_j$.

Intuition: Note that when the shares of the private input (say a) are over an odd ring (such as after using Π_{SC}), the MSB computation can be converted into an LSB computation. More precisely, over an odd ring, $\mathsf{MSB}(a) = \mathsf{LSB}(y)$, where $y = 2a$. Now, P_2 assists in computation of shares of $\mathsf{LSB}(y)$ as follows: P_2 picks a random integer $x \in \mathbb{Z}_{L-1}$ and sends shares of x over $\mathbb{Z}_{L-1}$ and shares of $x[0]$ over $\mathbb{Z}_L$ to P_0, P_1. Next, P_0, P_1 compute shares of $r = y + x$ and reconstruct r by exchanging shares. We note that $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \mathsf{wrap}(y, x, L-1)$ over an odd ring. Also, $\mathsf{wrap}(y, x, L-1) = (x > r)$, which can be computed using comparison protocol Π_{PC}. To enable this, P_2 also secret shares $\{x[i]\}_{i \in [\ell]}$ over $\mathbb{Z}_p$. Steps 6-10 compute the equation $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus (x > r)$ by using the arithmetic equation for xor computation (note that $x \oplus r = x + r - 2xr$; when one of x or y is public and known to both P_0 and P_1, then this computation can be done over the shares locally. When both are private and kept as shares, this computation is done using one call to multiplication (Step 9 in the protocol).).

Protocol	Rounds	Communication
$\mathsf{MatMul}_{m,n,v}$	2	$2(2mn + 2nv + mv)\ell$
$\mathsf{MatMul}_{m,n,v}$ (with PRF)	2	$(2mn + 2nv + mv)\ell$
SelectShare	2	5ℓ
PrivateCompare	1	$2\ell \log p$
ShareConvert	4	$4\ell \log p + 6\ell$
Compute MSB	5	$4\ell \log p + 13\ell$

Table 1. Round & communication complexity of *building blocks*.

4.6 Overheads of supporting protocols

The communication and round complexity of our supporting protocols is provided in Table 1. $\mathsf{MatMul}_{m,n,v}$ denotes matrix multiplication of an $m \times n$ matrix with an $n \times v$ matrix. The first row states the complexity of $\mathsf{MatMul}_{m,n,v}$ using secure Beaver triplets. In our implementation, we generate the triplets using PRFs as follows: P_0 and P_2 share a PRF key and use it to generate $\langle A \rangle_0^L, \langle B \rangle_0^L, \langle C \rangle_0^L$ locally. Similarly, P_1 and P_2 share a PRF key and use it to generate $\langle A \rangle_1^L, \langle B \rangle_1^L$ locally. Now, P_2 sets $\langle C \rangle_1^L = \mathsf{Reconst}^L(\langle A \rangle_0^L, \langle A \rangle_1^L) \cdot \mathsf{Reconst}^L(\langle B \rangle_0^L, \langle B \rangle_1^L) - \langle C \rangle_0^L$ and send to P_1. This reduces the communication of multiplication by half. All other complexities are for single elements and use this optimized version of multiplication.

5 Main Protocols

In this section, we describe all our main protocols for functionalities such as linear layer, derivative of ReLU, ReLU, division needed for normalization during training, Maxpool and its derivative. We maintain the invariant that parties P_0 and P_1 begin with "fresh" shares of input value (over $\mathbb{Z}_L, L = 2^{64}$) and output a "fresh" share of the output value (again over $\mathbb{Z}_L$) at the end of the protocol. Party P_2 takes the role of "assistant" in all protocols and has no input or output.

7 In the corner case when $r = 0$, both P_0 and P_1 set the output of Π_{PC} to be 1 and execute it. This is similar to the other corner case discussed in Section 4.3.

8 Most significant bit of a number is defined as the value of the leftmost bit in the bit representation.

5.1 Linear and Convolutional Layer

We note that a linear (or fully connected) layer in a neural network is exactly a matrix multiplication. Similarly, a convolutional layer can also be expressed as a (larger) matrix multiplication. As an example the 2-dimensional convolution of a 3×3 input matrix X with a kernel K of size 2×2 can be represented by the matrix multiplication shown below.

$$\mathsf{Conv2d}\left(\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}\right) = \begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{bmatrix} \times \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$$

For a generalization, see e.g. [4] for an exposition on convolutional layers. Hence both these layers can be directly implemented using Algorithm 1 from Section 4.

5.2 Derivative of ReLU

Algorithm 6 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\text{DReLU}}$ that computes the derivative of ReLU, denoted by ReLU'. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}'(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}'(a) = 1$ if $\mathsf{MSB}(a) = 0$, else $\mathsf{ReLU}'(a) = 0$.

Algorithm 6 ReLU', $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$, respectively.
Output: P_0, P_1 get $\langle \mathsf{ReLU}'(a) \rangle_0^L$ and $\langle \mathsf{ReLU}'(a) \rangle_1^L$.
Common Randomness: P_0, P_1 hold random shares of 0 over $\mathbb{Z}_L$, denoted by u_0 and u_1 resp.

1: For $j \in \{0,1\}$, parties P_j computes $\langle c \rangle_j^L = 2\langle a \rangle_j^L$.
2: P_0, P_1, P_2 run $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ with P_0, P_1 having inputs $\langle c \rangle_j^L$ & $\langle c \rangle_1^L$ & P_0, P_1 learn $\langle y \rangle_0^{L-1}$ & $\langle y \rangle_1^{L-1}$, resp.
3: P_0, P_1, P_2 run $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle y \rangle_j^{L-1}$ & P_0, P_1 learn $\langle \alpha \rangle_0^L$ & $\langle \alpha \rangle_1^L$, resp.
4: For $j \in \{0,1\}$, P_j outputs $\langle \gamma \rangle_j^L = j - \langle \alpha \rangle_j^L + u_j$.

Intuition: As is clear from the function ReLU' itself, the protocol computes the shares of $\mathsf{MSB}(a)$ and flips it to compute $\mathsf{ReLU}'(a)$. Recall that the protocol Π_{MSB} expects shares of a over $\mathbb{Z}_{L-1}$. Hence, we need to convert shares over $\mathbb{Z}_L$ to fresh shares over $\mathbb{Z}_{L-1}$ of the same value. Recall that for correctness of the share convert protocol, we require that value is not equal to $L-1$. This is ensured by first computing shares of $c = 2a$ and then calling Π_{SC}. We ensure[9] that $\mathsf{ReLU}'(a) = \mathsf{ReLU}'(c)$ by requiring that $a \in [0, 2^k) \cup (2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$.

5.3 ReLU

Algorithm 7 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\text{ReLU}}$ that computes $\mathsf{ReLU}(a)$. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}(a) = a$ if $\mathsf{MSB}(a) = 0$, else 0. That is, $\mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$.

Algorithm 7 ReLU, $\Pi_{\mathsf{ReLU}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$, respectively.
Output: P_0, P_1 get $\langle \mathsf{ReLU}(a) \rangle_0^L$ and $\langle \mathsf{ReLU}(a) \rangle_1^L$.
Common Randomness: P_0, P_1 hold random shares of 0 over $\mathbb{Z}_L$, denoted by u_0 and u_1 resp.

1: P_0, P_1, P_2 run $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle a \rangle_j^L$ and P_0, P_1 learn $\langle \alpha \rangle_0^L$ and $\langle \alpha \rangle_1^L$, resp.
2: P_0, P_1, P_2 call $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \alpha \rangle_j^L, \langle a \rangle_j^L)$ and P_0, P_1 learn $\langle c \rangle_0^L$ and $\langle c \rangle_1^L$, resp.
3: For $j \in \{0,1\}$, P_j outputs $\langle c \rangle_j^L + u_j$.

Intuition: Our protocol implements the above relation by using one call each to Π_{DReLU} and Π_{MatMul}. Note that Π_{MatMul} is invoked for multiplying two matrices of dimension 1×1 (or just one integer multiplication).

5.4 Division

We discuss our 3-party protocol Algorithm 8 realizing the functionality $\mathcal{F}_{\text{DIV}}$. Parties P_0, P_1 hold shares of x and y over $\mathbb{Z}_L$. At the end of the protocol, parties P_0, P_1 hold shares of $\lfloor x/y \rfloor$ over $\mathbb{Z}_L$ when $y \neq 0$.
Intuition: Our protocol implements long division where the quotient is computed bit-by-bit sequentially starting from the most significant bit. In each iteration, we compute the current dividend by subtracting the correct multiple of the divisor. Then we compare the current dividend with a multiple of the divisor ($2^i y$ in round

9 This essentially means that the absolute value of a is not very large, and in particular not larger than 2^k. This is not a limitation in any of the ML applications that we work with.

Algorithm 8 Division: $\Pi_{\mathsf{DIV}}(\{P_0, P_1\}, P_2)$

Input: P_0, P_1 hold $(\langle x\rangle_0^L, \langle y\rangle_0^L)$ and $(\langle x\rangle_1^L, \langle y\rangle_1^L)$, resp.
Output: P_0, P_1 get $\langle x/y\rangle_0^L$ and $\langle x/y\rangle_1^L$.
Common Randomness: $P_j, j \in \{0,1\}$ hold ℓ shares 0 over $\mathbb{Z}_L$ denoted by $w_{i,0}$ and $w_{i,1}$ for all $i \in [\ell]$ resp. They additionally also hold another share of 0 over $\mathbb{Z}_L$, denoted by s_0 and s_1.

1: Set $u_\ell = 0$ and for $j \in \{0,1\}$, P_j holds $\langle u_\ell\rangle_j^L$ (through the common randomness).
2: **for** $i = \{\ell - 1, \ldots, 0\}$ **do**
3: $P_j, j \in \{0,1\}$ compute $\langle z_i\rangle_j^L = \langle x\rangle_j^L - \langle u_{i+1}\rangle_j^L - 2^i\langle y\rangle_j^L + w_{i,j}$.
4: P_0, P_1, P_2 run $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle z_i\rangle_j^L$ and P_0, P_1 learn $\langle \beta_i\rangle_0^L$ and $\langle \beta_i\rangle_1^L$, resp.
5: P_0, P_1, P_2 call $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i\rangle_j^L, \langle 2^i y\rangle_j^L)$ and P_0, P_1 learn $\langle v_i\rangle_0^L$ and $\langle v_i\rangle_1^L$, resp.
6: $P_j, j \in \{0,1\}$ compute $\langle k_i\rangle_j^L = 2^i \cdot \langle \beta_i\rangle_j^L$.
7: For $j \in \{0,1\}$, P_j computes $\langle u_i\rangle_j^L = \langle u_{i+1}\rangle_j^L + \langle v_i\rangle_j^L$.
8: **end for**
9: For $j \in \{0,1\}$, P_j outputs $\langle q\rangle_j^L = \sum_{i=0}^{\ell-1}\langle k_i\rangle_j^L + s_j$.

i). Depending on the output of the comparison, i^{th} bit of the quotient is 0 or 1. This comparison can be written as a comparison with 0 and hence can be computed using a single call to Π_{DReLU}. We use this selection bit to select between 0 and 2^i for quotient and 0 and $2^i y$ for what to subtract from dividend. This selection can be implemented using Π_{MatMul} (similar to ReLU computation). Hence, division protocol proceeds in iterations and each iteration makes one call to Π_{DReLU} and one call[10] to Π_{MatMul}.

5.5 Maxpool

Algorithm 9 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{MAXPOOL}}$ to compute the maximum of n values. Parties P_0, P_1 hold shares of $\{x_i\}_{i\in[n]}$ over $\mathbb{Z}_L$ and end up with fresh shares of $\mathsf{max}(\{x_i\}_{i\in[n]})$.
Intuition: The protocol implements the max algorithm that runs in $(n-1)$ sequential steps. We start with $\mathsf{max}_1 = x_1$. In step i, we compute the shares of $\mathsf{max}_i = \mathsf{max}(x_1, \ldots, x_i)$ as follows: We compute shares of $w_i = x_i - \mathsf{max}_{i-1}$. Then, we compute shares of $\beta_i = \mathsf{ReLU}'(w_i)$ that is 1 if $x_i \geq \mathsf{max}_{i-1}$ and 0 otherwise. Next, we use Π_{SS} to select between max_{i-1} and x_i using β_i to compute max_i. Note, that in a similar manner, we can also calculate the index of maximum value, i.e. k such that $x_k = \mathsf{max}(\{x_i\}_{i\in[n]})$. This is done in steps 6&7. Computing the index of max value is required while doing prediction as well as to compute the derivative of Maxpool activation function needed for backpropagaton during training.

Algorithm 9 Maxpool $\Pi_{\mathsf{MP}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x_i\rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i\rangle_1^L\}_{i\in[n]}$, resp.
Output: P_0, P_1 get $\langle z\rangle_0^L$ and $\langle z\rangle_1^L$, resp., where $z = \mathsf{Max}(\{x_i\}_{i\in[n]})$.
Common Randomness: P_0 and P_1 hold two shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1 and v_0 and v_1.

1: For $j \in \{0,1\}$, P_j sets $\langle \mathsf{max}_1\rangle_j^L = \langle x_1\rangle_j^L$ and $\langle \mathsf{ind}_1\rangle_j^L = j$.
2: **for** $i = \{2, \ldots, n\}$ **do**
3: For $j \in \{0,1\}$, P_j computes $\langle w_i\rangle_j^L = \langle x_i\rangle_j^L - \langle \mathsf{max}_{i-1}\rangle_j^L$
4: P_0, P_1, P_2 call $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle w_i\rangle_j^L$ and P_0, P_1 learn $\langle \beta_i\rangle_0^L$ and $\langle \beta_i\rangle_1^L$, resp.
5: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i\rangle_j^L, \langle \mathsf{max}_{i-1}\rangle_j^L, \langle x_i\rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{max}_i\rangle_0^L$ and $\langle \mathsf{max}_i\rangle_1^L$, resp.
6: For $j \in \{0,1\}$, P_j sets $\langle k_i\rangle_j^L = j \cdot i$.
7: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i\rangle_j^L, \langle \mathsf{ind}_{i-1}\rangle_j^L, \langle k_i\rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{ind}_i\rangle_0^L$ and $\langle \mathsf{ind}_i\rangle_1^L$, resp.
8: **end for**
9: For $j \in \{0,1\}$, P_j outputs $(\langle \mathsf{max}_n\rangle_j^L + u_j, \langle \mathsf{ind}_n\rangle_j^L + v_j)$.

5.6 Derivative of Maxpool

The derivative of the Maxpool function (functionality $\mathcal{F}_{\text{DMAXPOOL}}$) is defined as the unit vector with a 1 only in the position with the maximum value. Here, we describe the more efficient Algorithm 10 that works for the special (and often-used) case of 2×2 Maxpool, where $n = 4$. In general, this algorithm works when n divides L. For the more general case, we provide an algorithm in Appendix D.

10 Note that multiplication with 2^i can be done locally.

Algorithm 10 Efficient Derivative of $n_1 \times n_2$ Maxpool $\Pi_{\mathsf{n_1xn_2DMP}}(\{P_0, P_1\}, P_2)$ with $n \mid L$ and $n = n_1 n_2$:

Input: P_0, P_1 hold $\{\langle x_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i \rangle_1^L\}_{i\in[n]}$, resp.

Output: P_0, P_1 get $\{\langle z_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle z_i \rangle_1^L\}_{i\in[n]}$, resp., where $z_i = 1$, when $x_i = \mathsf{Max}(\{x_i\}_{i\in[n]})$ and 0 otherwise.

Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L^n$ denoted by U_0 and U_1 and a random $r \in \mathbb{Z}_L$.

1: P_0, P_1, P_2 call $\mathcal{F}_{\text{MAXPOOL}}$ with $P_j, j \in \{0,1\}$ having input $\{\langle x_i \rangle_j^L\}_{i\in[n]}$, to obtain $\langle \mathsf{ind}_n \rangle_j^L$ resp. (from the second part of $\mathcal{F}_{\text{MAXPOOL}}$'s output).

2: P_0 sends $\langle k \rangle_0^L = \langle \mathsf{ind}_n \rangle_0^L + r$ to P_2, while P_1 sends $\langle k \rangle_1^L = \langle \mathsf{ind}_n \rangle_1^L$ to P_2.

3: P_2 computes $t = \mathsf{Reconst}^L(\langle k \rangle_0^L, \langle k \rangle_1^L)$, computes $k = t \bmod n$ and creates shares of E_k, denoted by $\langle E \rangle_0^L$ and $\langle E \rangle_1^L$, and sends the shares to P_0 and P_1 resp.

4: P_0 and P_1 locally "cyclic-shift" their shares by $g = r \bmod n$. That is, let $\langle E \rangle_j^L = (\langle E_0 \rangle_j^L, \langle E_1 \rangle_j^L, \cdots, \langle E_{n-1} \rangle_j^L)$. P_j computes $\langle D \rangle_j^L$ as $(\langle E_{(-g \bmod n)} \rangle_j^L, \langle E_{(1-g \bmod n)} \rangle_j^L, \cdots, \langle E_{(n-1-g \bmod n)} \rangle_j^L)$.

5: $P_j, j \in \{0,1\}$ outputs $\langle D \rangle_j^L + U_j$.

Intuition: The key observation behind this protocol is that when n divides L (i.e., $n \mid L$), we have that $a \bmod n = (a \bmod L) \bmod n$. The first step that P_0 and P_1 run is Π_{MP} that gives them shares of the index $\mathsf{ind} \in [n]$ with the maximum value. These shares are over L and must be converted into shares of the unit vector E_{ind} which is a length n vector with 1 in position ind and 0 everywhere else. P_0 and P_1 share a random $r \in \mathbb{Z}_n$ and have P_2 reconstruct $k = (\mathsf{ind} + r) \bmod n$. P_2 then creates shares of E_k and sends the shares back to P_0 and P_1 who "left-shift" these shares by r to obtain shares of E_{ind}. This works because $a \bmod n = (a \bmod L) \bmod n$ is true when $n \mid L$.

5.7 End-to-end Protocols

Our main protocols can be easily put together to execute training on a wide class of neural networks. For example, consider Network A, 3-layer neural network from SecureML that consists of a fully connected layer, followed by a ReLU, followed by another fully connected layer, followed by another ReLU, followed by the function $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ (for further details on this network, we refer the reader to [33]). To implement this, we first invoke Π_{MatMul}, followed by Π_{ReLU}, then again followed by Π_{MatMul} and Π_{ReLU} and finally we invoke Π_{DIV} to compute $\mathsf{ASM}(\cdot)$[11]. Backpropagaton is computed by making calls to Π_{MatMul} as well and Π_{DReLU} with appropriate dimensions[12]. Similarly, we can also do a general convolutional neural network with other activations such as Maxpool. We remark that we can put together these protocols easily since our protocols all maintain the invariant that parties begin with arithmetic shares of inputs and complete the protocol with arithmetic shares of the output.

6 Communication and Rounds

The round and communication complexity of our main protocols are presented in Table 2. The function $\mathsf{Linear}_{m,n,v}$ denotes a matrix multiplication of dimension $m \times n$ with $n \times v$. $\mathsf{Conv2d}_{m,i,f,o}$ denotes a convolutional layer with input $m \times m$, i input channels, a filter of size $f \times f$, and o output channels. l_D denotes precision of bits. $\mathsf{Maxpool}_n$ and DMP_n denotes Maxpool and its derivative over n elements. For ReLU and DMP_n, the overheads in addition to DReLU and $\mathsf{Maxpool}_n$ respectively are presented as these protocols are always implemented together in a neural network. All communication is measured for $\ell-$bit inputs and p denotes the field size (which is 67 in our case). All of the complexities are presented using the optimized complexity of multiplication that used PRFs for correlated randomess.

Our gains mainly come from the secure evaluation of non-linear functions such as ReLU and Maxpool and their derivatives. Prior works such as SecureML [33], MiniONN [30], Gazelle [26], etc took a garbled circuit-based approach to evaluate these functions – i.e., after

Protocol	Rounds	Communication
$\mathsf{Linear}_{m,n,v}$	2	$(2mn + 2nv + mv)\ell$
$\mathsf{Conv2d}_{m,i,f,o}$	2	$(2m^2f^2i + 2f^2oi + m^2o)\ell$
DReLU	8	$8\ell \log p + 19\ell$
ReLU (after DReLU)	2	5ℓ
NORM(l_D) or DIV(l_D)	$10l_D$	$(8\ell \log p + 24\ell)l_D$
$\mathsf{Maxpool}_n$	$9(n-1)$	$(8\ell \log p + 29\ell)(n-1)$
DMP_n (after Maxpool)	2	$2(n+1)\ell$

Table 2. Round & communication complexity of *main protocols*.

11 $\mathsf{ASM}(\cdot)$ consists of a summation and a division. Summation is a local computation and does not require a protocol to be computed.

12 We note that Π_{DReLU} is called as part of Π_{ReLU} in forward propagation and its value is stored for backpropagaton

completion of an arithmetic (linear) computation such as matrix multiplication, they ran a protocol to convert shares of intermediary values into an encoding suitable for garbled circuits. The non-linear function was then evaluated using the garbled circuit after which shares were once again converted back to be suitable for arithmetic computation. This approach leads to a multiplicative factor communication overhead proportional to the security parameter κ, as garbled circuits require communicating encodings proportional to κ, for every bit in the circuit. Overall, this leads to a communication complexity $> 768\ell$ for every $\ell-$ bit input [21]. As shown in [21], this cost of conversion to garbled circuits is $6\kappa\ell$, and all previous works incur this cost. In our approach, we provide new protocols to compute such non-linear activation functions, while continuing to retain arithmetic shares of the output values. For example, the ReLU protocol that we construct avoids paying κ multiplicative overhead and has communication complexity of $8\ell \log p + 24\ell$, which is approximately 88ℓ (when $p = 67$ as is in our setting). This leads to $> 8\times$ improvement in the communication complexity of the protocols for non-linear functions.

7 Evaluation

7.1 System Details

We test our prototype by running experiments over Amazon EC2 c4.8x large instances in two environments, respectively modeling a LAN and WAN setting.

- **LAN setting.** We use 3 Amazon EC2 c4.8xlarge machines running Ubuntu in the same region. The average bandwidth measured was 625MB/s and the average ping time was 0.22ms.
- **WAN setting.** In the WAN setting, we rent machines in different geographical regions with the same machine specifications as in the LAN setting. The average bandwidth measured was 40MB/s and the average ping time was 58ms.

Our system is implemented in about 7400 lines of C++ code with the use of standard libraries. We use the Eigen Library [1] for faster matrix multiplications. The ring is set to $\mathbb{Z}_{2^{64}}$ and we use the `uint64_t` native C++ datatype for all variables. Our source code is available at https://www.github.com/snwagh/securenn-public.git.

Number encoding. Typical neural networks work over floating point numbers. As observed by all prior works, to make them compatible with efficient cryptographic techniques, they must be encoded into fixed-point form. We use the methodology from [33] to support fixed-point arithmetic in an integer ring (described in Appendix B). The fixed-point numbers have 13 bits in their fractional part (cleartext training to get accuracy numbers is also done with these parameters).

7.2 Summary of experiments

We develop a prototype of SecureNN. We test the performance of our protocols by training 3 different neural networks over the MNIST dataset [3]. We also evaluate SecureNN on secure inference benchmarks in Section 7.5. Finally, in Section 7.6, we present microbenchmarks that measure the performance of various sub-protocols implemented in SecureNN such as Linear Layer, Convolutional Layer, ReLU and Maxpool (and its derivatives) that enables the estimation of the performance cost of other networks using the above functions.

For secure training, we run multiple iterations (10) and take the average - for each iteration, we measured the time for 10 forward-backward passes and used that to extrapolate the numbers for 15 epochs (7000 iterations). Secure inference timings are also averaged over 10 iterations. The learning rate is 2^{-5} in all experiments, except in the SecureML [33] network, where we retain their learning rate of 2^{-7}. In all our experiments, we report overall execution time (and do not split execution time into an offline, data independent phase, and an online, data dependent phase) and treat the same as online time as well. Our experiments show that our total execution times are better than even just the online times of previous works. If we split our work (e.g., triplet generation for multiplication) to an offline phase, our online improvements would be even better.

7.3 Neural Networks

For benchmarking and comparison, we consider four neural network architectures performing training/inference over the MNIST dataset [3] for handwritten digit recognition[13]. Network-A is a 3-layer DNN from [33], Network-B is a 4-layer CNN from [30, 31], Network-C is a 4-layer CNN from [27], and Network-D

13 This dataset has 60,000 training samples of handwritten digits. Each image is a 28-by-28 pixel square, with each pixel represented using 1 byte. The inference set contains 10,000 images.

is a 3-layer CNN from [31, 36]. We use these networks for training as well as inference and describe them in further detail in Appendix C.

7.4 Secure Training

We evaluate our protocols for secure training in both the LAN and WAN settings over the networks A, B, and C listed above. In many cases, the networks we train, achieve more than 99% accuracy for inference (on test dataset). We remark that we are the first work to show the feasibility of secure training on large and complex NNs such as CNNs that achieve high levels of accuracy. We vary the epochs between 5 and 15 for all networks except Network A which does not achieve good accuracy for smaller epochs and vary the batch size between 4 and 128 for networks B and C. Table 3 presents a summary of our results in the LAN/WAN setting as a function of the number of epochs for training (batch size fixed to 128), while Table 4 presents the results when the batch size is varied and the number of epochs is fixed to 5.

	Epochs	Accuracy	LAN (hours)	WAN (hours)
A	15	93.4%	1.03	7.83
B	5	97.94%	5.8	17.99
	10	98.05%	11.6	35.99
	15	98.77%	17.4	53.98
C	5	98.15%	9.98	30.66
	10	98.43%	19.96	61.33
	15	99.15%	29.95	91.99

Table 3. Secure training execution times for batch size 128.

	Batch size	Accuracy	LAN (hours)	WAN (hours)
B	4	99.15%	9.98	112.71
	16	98.99%	8.34	36.46
	128	97.94%	5.8	17.99
C	4	99.01%	18.31	123.96
	16	99.1%	13.43	46.2
	128	98.15%	9.98	30.66

Table 4. Secure training execution times for 5 epochs.

Comparison with prior work. The only prior work to consider neural network training was SecureML [33] that considers Network A only. They give implementations for both 2 and 3-server settings on similar hardware and network settings – we quote experimental numbers from their paper. We provide a comparison of our protocols with their work in Table 5. In the LAN setting, our protocol is roughly 6.8× faster than their 3 party protocol and 79× faster than their 2 party protocol. In the WAN setting, our improvements are even more dramatic and we get an improvement of 553× over the 2-party protocol[14]. Furthermore, SecureML split their times into a slow (data independent) offline phase and a faster (data dependent) online phase. Even comparing only their online time with our overall 3PC time, we obtain an improvement of 1.16× over their 2PC and a 2.7× improvement over their 3PC (their 3PC trades off some offline cost with a larger online cost).

	Framework	LAN (hr)			WAN (hr)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML 2PC	80.5	1.2	81.7	4277	59	4336
	SecureML 3PC	4.15	2.87	7.02	-	-	-
	SecureNN	0	1.03	1.03	0	7.83	7.83

Table 5. Training time comparison for Network A for batch size 128 and 15 epochs with SecureML [33].

	Framework	Runtime (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML	4.7	0.18	4.88	-	-	-
	SecureNN	0	0.043	0.043	0	2.1	2.1
B	MiniONN	3.58	5.74	9.32	20.9	636.6	657.5
	Gazelle	0.481	0.33	0.81	47.5	22.5	70.0
	SecureNN	0	0.13	0.13	0	8.86	8.86
C	SecureNN	0	0.23	0.23	0	18.94	18.94
D	DeepSecure	-	-	9.67	-	-	791
	Chameleon 3PC	1.34	1.36	2.7	7.8	5.1	12.9
	Gazelle	0.15	0.05	0.20	5.9	2.1	8.0
	SecureNN	0	0.076	0.076	0	4.05	4.05

Table 6. Single image inference time comparison of various protocols in the LAN setting.

7.5 Secure Inference

We also evaluate our protocols for the task of secure inference for the networks A, B, C and D. These

	LAN (s)		WAN (s)		Comm (MB)	
Batch size →	1	128	1	128	1	128
A	0.043	0.38	2.43	2.79	2.1	29
B	0.13	7.18	3.93	21.99	8.86	1066
C	0.23	10.82	4.08	30.45	18.94	1550
D	0.076	2.6	3.06	8.04	4.05	317.7

Table 7. Prediction timings for batch size 1 vs 128 for SecureNN on Networks A-D over MNIST.

14 SecureML do not provide numbers for their 3-party protocol in the WAN setting.

networks can either be a result of secure training using the 3PC protocol and are secret shared between P_0 and P_1, or a trained model can be secret shared between P_0 and P_1 at the beginning of the protocol.

Comparison with prior work. A sequence of previous works have considered a single secure inference in the LAN setting for various networks. Table 6 summarizes our comparison with state-of-the-art secure inference protocols. Networks-A and B were considered in SecureML [33], MiniONN [30], and Gazelle [26] using different techniques for secure computation. All these works used similar hardware and network settings as our LAN experiments and we quote experimental numbers from the respective papers.

Each of these works split their computation into an input independent offline phase and an input dependent online phase. In our protocols, we do not do this split and count all cost as online cost – hence, the offline cost is 0. Our protocols in the 3PC setting achieve roughly 3× improvement in small networks that have a small number of non-linear operations (such as Network D) and between 6×-113× improvements in some larger networks. In fact, in most cases, especially for realistic size networks, our total time is lower than the online time of previous best protocols (ignoring the offline time). We are the first to evaluate on Network C (which is considerably larger in size) and the table shows our runtime and communication. Finally, for network D, we also compare our protocols with the 3PC protocols in Chameleon [36]. This shows SecureNN improves on prior work by about 35×.

In all cases, our performance gains can be attributed to much better communication complexity of our protocols compared to previous works (see comparison in Table 6). In particular, as mentioned before, we avoid the use of garbled circuits for the non-linear activation functions such as ReLU. In all previous works, garbled circuits are the major factor in large communication.

Single vs Batch Prediction. Table 7 summarizes our results for secure inference over different networks for 1 prediction and batch of 128 predictions in both the LAN and WAN settings. Due to use of matrix-based Beaver triplets for secure multiplication protocol in linear and convolutional layers, and batching of communication, the time for multiple predictions grows sub-linearly. SecureML also did predictions for batch size 100 for Network A and took 14s and 143s in the LAN and the WAN settings, respectively. In contrast, we take only 0.38s and 2.79s for 128 predictions using 3PC protocol.

Comparison with ABY^3 [31]. ABY^3 considers a similar set-up as SecureNN but develop different techniques for matrix multiplication and non-linear operations. This results in protocols with different communication complexity with performance depending on the network architecture and hardware. For instance, ABY^3 requires 0.5MB of communication for inference on Network-A (SecureNN requires 2.1MB) while it requires 5.2MB of communication over Network-D (SecureNN requires 4.05MB). It would be interesting to see if their techniques can be combined with SecureNN to achieve the best of both worlds.

Protocol	Dimension	LAN (ms)	WAN (ms)	Comm. (MB)
Conv2d$_{m,f,i,o}$	$8, 5, 16, 50$	3.8	28.4	0.42
	$28, 3, 1, 20$	1.8	26.5	0.2
	$28, 5, 1, 20$	2.8	27.5	0.33
MatMul$_{m,n,v}$	$1, 100, 1$	0.33	25.2	0.0032
	$1, 500, 100$	4.8	29.4	0.81
	$784, 128, 10$	9.7	34.3	1.69
Maxpool	$8 \times 8 \times 50, 4 \times 4$	59.7	3062.2	2.23
	$24 \times 24 \times 16, 2 \times 2$	61.1	672.6	5.14
	$24 \times 24 \times 20, 2 \times 2$	62.6	685	6.43
DMP	$8 \times 8 \times 50, 4 \times 4$	1.9	51.6	0.18
	$24 \times 24 \times 16, 2 \times 2$	4.8	54.2	0.52
	$24 \times 24 \times 20, 2 \times 2$	4.9	55.2	0.65
DReLu	64×16	11.2	161.9	0.68
	128×128	109.8	288.7	10.88
	576×20	71.5	232.9	7.65
ReLU	64×16	0.42	25.3	0.04
	128×128	2.8	27.1	0.66
	576×20	2.5	26.6	0.46

Table 8. Microbenchmarks in LAN/WAN settings.

7.6 Microbenchmarks

Table 8 presents microbenchmark timings for various ML functionality protocols varied across different dimensions. All timings are average timings. The overheads for DMP and ReLU are additional over the costs of Maxpool and DReLU respectively as these pairs of protocols are always used together in training.

Acknowledgments

We thank Sai Lakshmi Bhavana Obbattu and Bhavana Kanukurthi for helpful discussions on the technical aspects of the paper. We thank Shruti Tople, Abhishek Bichhawat, and Tri Nguyen for their help with the implementation and Harshavardhan Simhadri and Rahul Sharma for their very useful pointers in making the code better. This work is supported in part by Army Research Office YIP award, National Science Foundation CIF-1617286 and CNS-1553437 grants.

References

[1] Eigen Library. http://eigen.tuxfamily.org/. Version: 3.3.3.

[2] Fixed-point data type. http://dec64.com. Last Updted: 2018-01-20.

[3] MNIST database. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-09-24.

[4] Stanford CS231n: Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks/.

[5] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (GDPR). *Official Journal of the European Union*, L119, May 2016.

[6] 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 CCS*, 2016.

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

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *ACM STOC*, 1988.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[10] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[11] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS '01, pages 136–, 2001.

[13] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[14] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, pages 182–199, 2010.

[15] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.cms.hhs.gov/hipaa/, 1996.

[16] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. https://eprint.iacr.org/2017/035.

[17] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *ACM STOC*, 1988.

[18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Crypto*, pages 34–64, 2018.

[19] Benny Chor and Eyal Kushilevitz. A zero-one law for boolean privacy. *SIAM J. Discrete Math.*, 4(1), 1991.

[20] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Homomorphic encryption and secure comparison. In *IJACT*, 2008.

[21] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[22] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. 2014.

[23] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *IACR Eurocrypt*, 2017.

[24] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[25] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *ACM STOC*, 1987.

[26] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *Usenix Security*, 2018.

[27] 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.

[28] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, 2017.

[29] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Annual International Cryptology Conference*, pages 36–54. Springer, 2000.

[30] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS*, 2017.

[31] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[32] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS*, 2015.

[33] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *ieee-oakland*, 2017.

[34] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *ACM CCS*, pages 801–812, 2013.

[35] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, 2007.

[36] 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 *AsiaCCS*, 2018.

[37] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321. ACM, 2015.

[38] Wagh, Sameer and Gupta, Divya and Chandran, Nishanth. SecureNN: 3-Party Secure Computation for Neural Network Training. https://eprint.iacr.org/2018/442.pdf, 2019.

[39] David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016, 2016.

[40] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *IEEE FOCS*, 1986.

[41] Xiangxin Zhu, Carl Vondrick, Charless Fowlkes, and Deva Ramanan. Do we need more training data? In *International Journal of Computer Vision*, 2016.

A Other Related Work

In recent years, privacy-preserving machine learning has received considerable research attention. We first discuss the most closely related works that consider neural network inference and training, and then provide an overview of other related works.

Neural Network Inference and Training. Perhaps the first work to consider secure neural network prediction was the work of Gilad-Barach *et al.* [24] who used homomorphic encryption techniques to provide secure prediction. For efficiency reasons, they approximated non-linear functions, such as the ReLU activation function to a quadratic function. Since this approximation results in loss in accuracy, there have been works that approximate ReLU using higher degree polynomials [16], but incur higher cost.

The work of SecureML [33] provided secure protocols for neural network training and prediction with non-linear activations, using a combination of arithmetic and Yao's garbled circuit techniques. They provided computational security against a single semi-honest adversary in both the 2 and 3-server models. The work of MiniONN [30] further optimized the protocols of SecureML [33] (specifically reducing the offline cost of matrix multiplications by increasing the online cost) for the case of prediction in the 2-server model. They also provided computational security against a semi-honest adversary. Concurrently and independently to this work, the works of Chameleon [36] and Gazelle [26] provide secure inference protocols in the 3-server and 2-server models, respectively. Chameleon remove expensive oblivious transfer protocols (needed for secure multiplications) by using the third party as a dealer, while Gazelle focusses on making the linear layers (such as matrix multiplication and convolution) more communication efficient by providing specialized packing schemes for additively homomorphic encryption schemes. Both these works are also computationally secure against one semi-honest adversary. All of the above protocols [26, 30, 33, 36] use garbled circuits for non-linear activations.

In contrast to all the above works, we provide protocols for non-linear activation functions by avoiding garbled circuits and dramatically reduce their communication complexity. Additionally, protocols in SecureNN enjoy information-theoretic security (barring computational assumptions for implementations) as well as provide privacy against malicious adversaries.

Other related works. Bost *et al.* [11] propose a number of building block functionalities to perform secure inference for linear classifiers, decision trees and naive bayes in the two-party setting. Later, [39] gave an improved protocol for decision trees. Perhaps the first work to consider secure training was that of Lindell and Pinkas [29] who provided algorithms to execute decision tree based training over shared data. Nikolaenko et al. [34] implemented a secure matrix factorization to train a movie recommender system. Shokri and Smatikov [37] considered the problem of privacy in neural network training when data is horizontally partitioned. Here, the parties run the training on their data individually, and exchange the changes in coefficients obtained during training – the goal is to minimize leakage and provide privacy to users using various techniques such as differential privacy [22].

B Arithmetic operations on shared decimal numbers

In order for neural network algorithms to be compatible with cryptographic applications, they must typically be encoded into integer form (most neural network algorithms work over floating point numbers). Now, decimal arithmetic must be performed over these values in an integer ring which requires careful detail. We follow [33] and describe details below. We use fixed point arithmetic to perform all computations. In other words, all numbers are represented as integers in the native C++ datatype `uint64_t`. We use a precision of $l_D = 13$ bits for representing numbers. For instance, an integer 2^{15} in this encoding corresponds to the float 4 and an integer $2^{64} - 2^{13}$ corresponds to a float -1. Since we use unsigned integers for encoding, $\mathsf{ReLU}(\cdot)$ compares its argument with 2^{63}. Such encoding is gaining popu-

larity in the systems community with the introduction of fixed-point data types [2].

To perform decimal arithmetic in an integer ring, we use the same solution as is used in [33]. Addition of two fixed point decimal numbers is straightforward. To perform multiplication, we multiply the two decimal numbers and *truncate* the last l_D bits of the product. Theorem 1 in [33] shows that this above truncation technique also works over shared secrets (2-out-of-2 shares) i.e., the two parties can simply truncate their shares locally preserving correctness with an error of at most 1 bit with high probability. Denoting an arithmetic shift by $\Pi_{\mathsf{AS}}(a, \alpha)$, truncation of shares i.e., dividing shares by a power of 2 is described in Algorithm 11. We refer the reader to [33] for further details.

Algorithm 11 Truncate $\Pi_{\mathsf{Truncate}}(\{P_0, P_1\})$:

Input: P_0 & P_1 hold an positive integer α and $\langle X \rangle_0^L$ & $\langle X \rangle_1^L$ resp.

Output: P_0 gets $\langle X/2^\alpha \rangle_0^L$ and P_1 gets $\langle X/2^\alpha \rangle_1^L$.

1: P_0 computes $\Pi_{\mathsf{AS}}(\langle X \rangle_0^L, \alpha)$.
2: P_1 computes $-\Pi_{\mathsf{AS}}(-\langle X \rangle_1^L, \alpha)$.

C Neural Networks

Network A. This is the Deep Neural Network from [33] which is a 3-layer network comprising of only fully connected (linear) layers and uses ReLU as the activation function. During training of this network, $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ is applied to the output of the last layer to convert the output values into a probability distribution before doing the backpropogation. The backpropogation updates the weights appropriately making use of derivative of the activation function (in this case $\mathsf{ReLU}'(x)$, which is defined to be 1 if $x > 0$ and 0 otherwise) and matrix multiplication. This network, after training for 15 epochs, has a prediction accuracy of 93.4% as illustrated in [33].

Network B. Next is the Convolutional Neural Network from [30]; while [30] used this network for prediction, we use the network to train over the MNIST dataset. This is a 4-layer convolutional neural network that has the following structure. First is a 2-dimensional convolutional layer with 1 input channel, 16 output channels and a 5×5 filter. The activation functions following this layer are ReLU, followed by a 2×2 Maxpool. The second layer is a 2-dimensional convolutional layer with 16 input channels, 16 output channels and another 5×5 filter. The activation functions following this layer are once again ReLU and a 2×2 Maxpool. The third layer is an 256×100 fully-connected layer. The next activation function is ReLU. The final layer is a 100×10 linear layer and this is normalized using $\mathsf{ASM}(\cdot)$ to get a probability distribution. The loss function is cross entropy and stochastic gradient descent is used to minimize loss. Backpropogation equations are computed appropriately. We show that this network, after training for 15 epochs, provides an inference accuracy of 98.77% on the MNIST dataset.

Network C. Finally, we also run our protocols over a (standard) LeNet network [27], which is a larger version of the network from [30]. This is a 4-layer convolutional neural network with similar structure as above but more number of output channels and bigger linear layers. First layer is a 2-dimensional convolutional layer with 1 input channel, 20 output channels and a 5×5 filter. The activation functions following this layer are ReLU, followed by a 2×2 Maxpool. The second layer is a 2-dimensional convolutional layer with 20 input channels, 50 output channels and another 5×5 filter. The activation functions following this layer are once again ReLU and a 2×2 Maxpool. The third layer is an 800×500 fully-connected layer. The next activation function is ReLU. The final layer is a 500×10 linear layer and this is normalized using $\mathsf{ASM}(\cdot)$ to get a probability distribution. We show that this network, after training for 15 epochs, has a prediction accuracy of 99.15%.

Network D. In addition to these networks for training, for the case of secure inference, we also consider a network from Chameleon [36] for comparison in the 3-party setting. This network's structure is as follows: the first layer is a 2-dimensional convolutional layer with a 5×5 filter, stride of 2, and 5 output channels. The activation function next is ReLU. The second layer is a fully connected layer from a vector of size 980 to a vector of size 100. Next is another ReLU activation function. The last layer is a fully connected layer from a vector of size 100 to a vector of size 10. Finally the $\mathsf{arg\ max}$ function is used to pick among the 10 values for predicting the digit. This network gives inference accuracy of 99%.

D Security Proofs

Here, we provide proofs of semi-honest simulation based security for a subset of our protocols and defer

the remaining to full version. If a protocol invokes another sub-protocol for a functionality $\mathcal{F}$, we prove the security by replacing the sub-protocol invocation with the corresponding functionality call. This refers to $\mathcal{F}$-hybrid model.

Private Compare

Lemma 2. *Protocol* $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ *in Algorithm 3 securely realizes* $\mathcal{F}_{\text{PC}}$ *when* $p > \ell + 2$.

Proof. We first prove correctness of our protocol, i.e., $\beta' = \beta \oplus (x > r)$. Define $x[i]$ as $x[i] := \mathsf{Reconst}^p(\langle x[i]\rangle_0^p, \langle x[i]\rangle_1^p) \in \{0,1\}$ for all $i \in [\ell]$. We treat x and r as ℓ bit integers and $x > r$ tells if x is greater[15] than r. Below, we do a case analysis on value of β.

Case $\beta = 0$. For correctness, we require $\beta' = (x > r)$. For each $i \in [\ell]$, define $w_i = \mathsf{Reconst}^p(\langle w_i\rangle_0^p, \langle w_i\rangle_1^p)$. Note that $w[i] = x[i] + r[i] - 2r[i]x[i] = x[i] \oplus r[i]$. For each $i \in [\ell]$, define $c_i = \mathsf{Reconst}^p(\langle c_i\rangle_0^p, \langle c_i\rangle_1^p)$. Note that $c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{\ell} w_k$. Let i^* be such that for all $i > i^*$, $x[i] = r[i]$ and $x[i^*] \neq r[i^*]$. We claim that the following holds:

- For all $i > i^*$, $c[i] = 1$. This is because both $r[i] - x[i]$ and $\sum_{k=i+1}^{\ell} w_k$ are 0.
- For $i = i^*$, if $x[i] = 1$, $c[i] = 0$, else $c[i] = 2$.
- For $i < i^*$, $c[i] > 1$. This is because $r[i] - x[i]$ is either 1 or -1 and $\sum_{k=i+1}^{\ell} w_k > 1$. For this step, we require that there is no wrap around modulo p, which is guaranteed by $p > \ell + 2$.

This proves that $x > r$ iff there exists a $i \in [\ell]$ such that $c[i] = 0$. Finally, the last step of multiplying with random non-zero s_i and permuting all the $s_i c_i$ preserves this characteristic. This condition is exactly what P_2 checks.

Case $\beta = 1$. For correctness, we require $\beta' = 1 \oplus (x > r) = (x \leq r)$. The last expression is equivalent to $x < (r+1)$ when $r \neq 2^\ell - 1$ and otherwise $x \leq r$ is always true. Note that $t = r + 1$. Now, similar to logic above, we compute $t > x$ when $r \neq 2^\ell - 1$. This condition is easy to check since r is known to both P_0 and P_1.

When $r = 2^\ell - 1$, we know that $\beta' = 1$. Also, $\beta' = 1$ iff there exists a unique i such that d_i is 0. Hence, the parties create a vector starting with 1 followed by $\ell - 1$ zeroes. Scaling by s_i and permutation creates a uniform vector with exactly one 0.

15 $x > r$ iff the leftmost bit where $x[i] \neq r[i]$, $x[i] = 1$.

Now we prove security of our protocol. First note that P_0 and P_1 receive no messages in the protocol and hence, our protocol is trivially secure against corruption of P_0 or P_1. Now, we have to simulate the messages seen by P_2 given P_2's output, namely β'. To do this, if $\beta' = 0$, pick $d_i \xleftarrow{\$} \mathbb{Z}_p^*$, for all $i \in [\ell]$. If $\beta' = 1$, then pick an $i^* \xleftarrow{\$} [\ell]$, set $d_{i^*} = 0$ with all other $d_i \xleftarrow{\$} \mathbb{Z}_p^*$. Now, compute $(\langle d_i\rangle_0^p, \langle d_i\rangle_1^p) \leftarrow \mathsf{Share}^p(d_i)$ and send $\langle d_i\rangle_j^p$ for all $i \in [\ell], j \in \{0,1\}$ as the message from P_j to P_2. This completes the simulation. To see that the simulation is perfect, observe that whether or not $\exists i^*$, with $d_{i^*} = 0$ depends only on β'. Additionally, when $\beta' = 1$, the index i^* where $d_{i^*} = 0$ is uniformly random in $[\ell]$ due to the random permutation π. Finally, the non-zero d_i values are uniform over $\mathbb{Z}_p^*$ since the s_i values are random in $\mathbb{Z}_p^*$. □

Compute MSB

Lemma 3. *Protocol* $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ *in Algorithm 5 securely realizes* $\mathcal{F}_{\text{MSB}}$ *in the* $(\mathcal{F}_{\text{PC}}, \mathcal{F}_{\text{MATMUL}})$*-hybrid model.*

Proof. First, we prove correctness of our protocol, i.e., $\alpha := \mathsf{Reconst}^L(\langle\alpha\rangle_0^L, \langle\alpha\rangle_1^L) = \mathsf{MSB}(a)$. As already mentioned, over an odd ring, the MSB computation can be reduced to LSB computation. More precisely, over an odd ring, $\mathsf{MSB}(a) = \mathsf{LSB}(y)$, where $y = 2a$. Hence, it suffices to compute $\mathsf{LSB}(2a)$.

In the protocol, $r = y + x(\mathsf{mod} L - 1)$. Hence, $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \mathsf{wrap}(y, x, L-1)$. Next, we note that $\mathsf{wrap}(y, x, L-1) = (x > r)$. First, P_0, P_1, P_2 compute $x > r$ as follows. They invoke Π_{PC} and its correctness ensures that P_2 learns $\beta' = \beta \oplus (x > r)$. Next, P_2 secret shares β' to P_0, P_1. Note that $\gamma = \beta' + \beta - 2\beta\beta' = \beta \oplus \beta' = (x > r) = \mathsf{wrap}(y, x, L-1)$. Next, similarly, $\delta = r[0] \oplus x[0]$. Then, $\theta = \gamma\delta$ and $\alpha = \gamma + \delta - 2\theta = \gamma \oplus \delta = \mathsf{LSB}(y) = \mathsf{MSB}(a)$.

Next, we prove security of our protocol. Parties P_0 and P_1 learn the following information: $2a + x$ (from Step 3), $\langle r\rangle_j^{L-1}$, $\{\langle x[i]\rangle_j^p\}_i$, $\langle x[0]\rangle_j^B$ (Step 1) and $\langle\beta'\rangle_j^B$ (Step 5). However, these are all fresh shares of these values and hence can be perfectly simulated by sending random fresh share of 0. Finally, P_j outputs a fresh share of $\mathsf{MSB}(a)$ as the share is randomized with u_j. The only information that P_2 learns is bit β'. However, $\beta' = \beta \oplus (r > c)$, where β is a random bit unknown to P_2. Hence, the distribution of β' is uniformly random from P_2's view and hence the information learned by P_2 can be perfectly simulated. □

Derivative of ReLU

Lemma 4. *Protocol* $\Pi_{\mathsf{DRELU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 6 securely realizes* $\mathcal{F}_{\text{DReLU}}$ *in the* $(\mathcal{F}_{\text{SC}}, \mathcal{F}_{\text{MSB}})-$*hybrid model for all* $a \in [0, 2^k] \cup [2^\ell - 2^k, 2^\ell - 1]$*, where* $k < \ell - 1$.

Proof. First, we prove the correctness of our protocol when $a \in [0, 2^k) \cup (2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$, i.e., $\gamma := \mathsf{Reconst}^L(\langle\gamma\rangle_0^L, \langle\gamma\rangle_1^L) = \mathsf{ReLU}'(a) = 1 \oplus \mathsf{MSB}(a)$, where a is the value underlying the input shares. Note that when a belongs to the range $[0, 2^k] \cup [2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$, $\mathsf{MSB}(a) = \mathsf{MSB}(2a) = \mathsf{MSB}(c)$. Also, it holds that $2a \neq L - 1$, and precondition of $\mathcal{F}_{\text{SC}}$ is satisfied. From correctness of $\mathcal{F}_{\text{SC}}$, $y := \mathsf{Reconst}^{L-1}(\langle y\rangle_0^{L-1}, \langle y\rangle_1^{L-1}) = 2a$. Next, from correctness of $\mathcal{F}_{\text{MSB}}$, $\alpha := \mathsf{Reconst}^L(\langle\alpha\rangle_0^L, \langle\alpha\rangle_1^L) = \mathsf{MSB}(y) = \mathsf{MSB}(2a)$. Finally, $\gamma = 1 - \alpha = 1 - \mathsf{MSB}(a)$ as required. Also, note that $\langle\gamma\rangle_j^L$ are fresh shares of γ since both parties locally add shares of 0 to randomize the shares.

For security, first see that P_2 learns no information from the protocol (as both $\mathcal{F}_{\text{SC}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\text{MSB}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learns a fresh share of $2a$ (over $\mathbb{Z}_{L-1}$) in Step 2 and a fresh share of $\alpha = \mathsf{MSB}(2a)$ in Step 3 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0. Finally, P_j outputs a fresh share of $\mathsf{ReLU}'(a)$ as the respective shares are randomized by u_j. □

ReLU

Lemma 5. *Protocol* $\Pi_{\mathsf{ReLU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 7 securely realizes* $\mathcal{F}_{\text{ReLU}}$ *in the* $(\mathcal{F}_{\text{DReLU}}, \mathcal{F}_{\text{MATMUL}})$*-hybrid model.*

Proof. First, we prove the correctness, i.e., $c := \mathsf{Reconst}^L(\langle c\rangle_0^L, \langle c\rangle_1^L) = \mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$, where a is the value underlying the input shares. It follows from correctness[16] of $\mathcal{F}_{\text{DReLU}}$ that $\alpha := \mathsf{Reconst}^L(\langle\alpha\rangle_0^L, \langle\alpha\rangle_1^L) = \mathsf{ReLU}'(a)$. Now from the correctness of $\mathcal{F}_{\text{MATMUL}}$ it follows that $c = \alpha \cdot a$.

For security, see that P_2 learns no information from the protocol (as both $\mathcal{F}_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learns a fresh share of $\alpha = \mathsf{ReLU}'(a)$ in Step 1 and a fresh share of αa (over $\mathbb{Z}_L$) in Step 2 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0. Finally, P_j outputs a fresh share of $\mathsf{ReLU}(a)$ as the respective shares are randomized by u_j. □

16 When we instantiate the functionality $\mathcal{F}_{\text{DReLU}}$ using protocol Π_{DReLU}, we would ensure that the conditions on range of input to Π_{DReLU} are met.

E Privacy against malicious adversary

In this section, we show that all our protocols described in Sections 4 and 5 as well as protocols for general neural networks obtained by putting these together satisfy stronger security requirement than semi-honest security, namely, privacy against a malicious server in the client-server model (formalized by [6]). As was already pointed out by Araki et al. [6], this can only be achieved when the servers receive no information about the output of the protocol. Formally, we show that, for any malicious server, for any two inputs of the honest clients (holding the data) the view of the server is indistinguishable.

First, intuitively, we show that views are identical with secure correlated randomness. This holds because in all our protocol, the incoming messages to a server are either a fresh share of a value or can be generated using a uniformly random value (e.g., incoming messages of P_2 in private-compare protocol). Thus, irrespective of what the adversary sends in each round, the view of a malicious server can be simulated using uniform randomness and is completely independent of the inputs being used by the clients.

Second, in the case when correlated randomness is generated using shared PRF keys, to argue security against malicious P_0, we rely on security of the PRF key shared between P_1, P_2 that is unknown to P_0. Using this, we show that incoming messages of P_0 are computationally close to uniform. It is critical that to argue security against a malicious P_0 we do not rely on security of PRF keys known to P_0, i.e. shared keys between P_0, P_1 or P_0, P_2. Hence, we do not need to use a malicious secure coin-tossing protocol to generate secure keys between an adversary and an honest server. We only rely on the security of the PRF key shared between two honest servers. Therefore, the exact same protocol gives privacy against a single malicious server. Similar arguments can be made to argue security against a malicious P_1 or malicious P_2.

F Remaining protocols and security proofs

Here, we provide the proofs for the remaining protocols described in Section 4 and Section 5.

3-party Matrix Multiplication

Lemma 6. *Protocol* $\Pi_{\mathsf{MalMul}}(\{P_0, P_1\}, P_2)$ *in Algorithm 1 securely realizes* $\mathcal{F}_{\text{MATMUL}}$.

Proof. Let Z_j be the output of the party P_j. For correctness we need to prove that i.e. $\mathsf{Reconst}^L(Z_0, Z_1) = X \cdot Y$. We calculate $Z_0 + Z_1 = \left(\langle X\rangle_0^L \cdot F + E \cdot \langle Y\rangle_0^L + \langle C\rangle_0^L + U_0\right) + \left(-E \cdot F + \langle X\rangle_1^L \cdot F + E \cdot \langle Y\rangle_1^L + \langle C\rangle_1^L + U_1\right) = -E \cdot F + X \cdot F + E \cdot Y + C = -(X - A) \cdot (Y - B) + X \cdot (Y - B) + (X - A) \cdot Y + A \cdot B = X \cdot Y$.

Security against corrupt P_2 is easy to see since it gets no message and only generates a fresh matrix Beaver triplet of correct dimensions. Now, we prove security against corruption of either P_0 or P_1. Party P_0 receives $\langle A\rangle_0^L, \langle B\rangle_0^L, \langle C\rangle_0^L$ and $\langle E\rangle_1^L, \langle F\rangle_1^L$. We note that all of these uniform random matrices because A, B are uniformly chosen and fresh shares are generated of A, B, C. Also, the final output of $P_j, j \in \{0, 1\}$ is a fresh random share of $X \cdot Y$ (as they have each been randomized by random matrix U_j) and contain no information about X and Y. □

Select Share

Lemma 7. *Protocol* $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ *in Algorithm 2 securely realizes* $\mathcal{F}_{\text{SS}}$ *in the* $\mathcal{F}_{\text{MATMUL}}$*−hybrid model.*

Proof. We first prove the correctness of our protocol, i.e., $z := \mathsf{Reconst}^L(\langle z\rangle_0^L, \langle z\rangle_1^L)$ is x when $\alpha = 0$ and y when α is 1. Note that $w = y - x$ and from correctness of $\mathcal{F}_{\text{MATMUL}}$, $c = \mathsf{Reconst}^L(\langle c\rangle_0^L, \langle c\rangle_1^L) = \alpha \cdot w = \alpha \cdot (y - x)$. And finally, $z = x + c = (1 - \alpha) \cdot x + \alpha \cdot y$. Hence, correctness holds.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ provides outputs only to P_0 and P_1). On the other hand, $P_j, j \in \{0, 1\}$ only learn fresh shares of the outputs in Step 2 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the output in Step 3 as they are randomized by u_j. □

Share Convert

Proof of Lemma 1: We have already seen correctness. To see the security, first observe that the only information that P_2 sees is $x = a + r$ (over $\mathbb{Z}_L$) and η'. Since $r \xleftarrow{\$} \mathbb{Z}_L$ and is not observed by P_2, we have that x is uniform over $\mathbb{Z}_L$ and so information sent to P_2 can be simulated by sampling $x \xleftarrow{\$} \mathbb{Z}_L$ and sending shares of x from P_j to P_2 for $j \in \{0, 1\}$. Next, η'' is a random bit not observed by P_2 and thus, η' is a uniform random bit to P_2. Hence, η' can be perfectly simulated.

Finally, the only information that P_0 and P_1 observe are fresh shares of the following values: $\forall i \in [\ell], x[i]$, δ, and η' that can be perfectly simulated by sharing 0. The outputs of P_0 and P_1 are fresh shares of a over $\mathbb{Z}_{L-1}$ as they are randomized using u_0 and u_1 respectively.

Division

Lemma 8. *Protocol* $\Pi_{\mathsf{DIV}}(\{P_0, P_1\}, P_2)$ *in Algorithm 8 securely realizes* $\mathcal{F}_{\text{DIV}}$ *in the* $(\mathcal{F}_{\text{DReLU}}, \mathcal{F}_{\text{MATMUL}})$*−hybrid model when* $y \neq 0$.

Proof. We first prove the correctness of our protocol, i.e., $q := \mathsf{Reconst}^L(\langle q\rangle_0^L, \langle q\rangle_1^L) = \lfloor x/y \rfloor$. Our protocol mimics the standard long division algorithm and proceeds in ℓ iterations. In the i^{th} iteration we compute the $q[i]$, the i^{th} bit of q starting from the most significant bit.

We will prove by induction and maintain the invariant: $\beta_i = q[i]$, $k_i = 2^i \beta_i$, $u_i = y \cdot \sum_{j=i}^{\ell-1} k_j$. Assume that invariant holds for $i > m$, then we will prove that it holds for $i = m$. Note that $z_m = (x - u_{m+1} - 2^m y)$. We note that β_m or $q[m]$ is 1 iff $x - u_{m+1} > 2^m y$, that is, when $\mathsf{ReLU}'(z_m) = 1$. By correctness[17] of $\mathcal{F}_{\text{DReLU}}$, $\beta_m = \mathsf{Reconst}^L(\langle \beta_m\rangle_0^L, \langle \beta_m\rangle_1^L) = \mathsf{ReLU}'(z_m)$. Next by correctness of $\mathcal{F}_{\text{MATMUL}}$, $k_m = \beta_m 2^m$ and $v_m = \beta_m \cdot 2^m y = k_m y$. Hence, $u_m = u_{m+1} + v_m = y \cdot \sum_{j=m}^{\ell-1} k_j$.

To argue security, first observe that P_2 learns no information from the protocol (as both $\mathcal{F}_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learn fresh shares of the outputs in Step 4, 5 and 6 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0

17 When we instantiate the functionality $\mathcal{F}_{\text{DReLU}}$ using protocol Π_{DReLU}, we would ensure that the conditions of Lemma 4 are met.

(over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the final output in Step 9 as they are randomized by s_j. □

Maxpool

Lemma 9. *Protocol* $\Pi_{\mathsf{MP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 9 securely realizes* $\mathcal{F}_{\text{MAXPOOL}}$ *in the* $(\mathcal{F}_{\text{DReLU}}, \mathcal{F}_{\text{SS}})-$*hybrid model.*

Proof. We first prove the correctness of our protocol, i.e., $\mathsf{max}_n := \mathsf{Reconst}^L(\langle \mathsf{max}_n \rangle_0^L, \langle \mathsf{max}_n \rangle_1^L)$ stores the maximum value of the elements $\{x_i\}_{i\in[n]}$ and $\mathsf{ind}_n := \mathsf{Reconst}^L(\langle \mathsf{ind}_n \rangle_0^L, \langle \mathsf{ind}_n \rangle_1^L)$ stores the index of maximum value.

We will prove this by induction and will maintain the invariant that max_i holds the value of $\mathsf{max}(x_1, \ldots, x_i)$ and ind_i holds a value of k s.t. $\mathsf{max}_i = x_k$. It is easy to see that this holds for $i = 1$. Suppose this holds for $i = m-1$. Then we will prove that it holds for $i = m$. In Step 3, we calculate $w_m = x_m - \mathsf{max}_{m-1}$. By correctness [18] of $\mathcal{F}_{\text{DReLU}}$, $\beta_m = \mathsf{ReLU}'(w_m)$. That is, $\beta_m = 1$ iff $x_m > \mathsf{max}_{m-1}$. Next, by correctness of $\mathcal{F}_{\text{SS}}$, max_m is max_{m-1} if $\beta_m = 0$ and x_m otherwise. In Step 6, we compute shares of $k_m = m$. In Step 7, by correctness of $\mathcal{F}_{\text{SS}}$, $\mathsf{ind}_m = \mathsf{ind}_{m-1}$ if $\beta_m = 0$ and m otherwise. This proves correctness.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$ provides outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learn fresh shares of the values $\beta_i, \mathsf{max}_i, \mathsf{ind}_i$ and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh shares of the final output in Step 9 as the respective shares are randomized by u_j and v_j. □

Derivative of Maxpool

We provide a proof of correctness and security of Algorithm 10 followed by the general case algorithm.

Lemma 10. $\Pi_{\mathsf{n}_1 \times \mathsf{n}_2 \mathsf{DMP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 10 securely realizes* $\mathcal{F}_{\text{DMAXPOOL}}$ *in the* $\mathcal{F}_{\text{MAXPOOL}}-$*hybrid model.*

Proof. Let k^* be the index of the maximum value and E_r denote the unit vector with 1 in the r^{th} position and 0 everywhere else. For correctness, we show that $\mathsf{Reconst}^L(\langle D \rangle_0^L + U_0, \langle D \rangle_1^L + U_1) = E_{k^*}$ in Algorithm 10.

From the correctness of $\mathcal{F}_{\text{MAXPOOL}}$, we have that P_0 and P_1 hold shares of ind_n (which is the index of the maximum value). P_2 receives $\langle \mathsf{ind}_n \rangle_0^L + r$ and $\langle \mathsf{ind}_n \rangle_1^L$ from P_0 and P_1 resp. and reconstructs $t = \mathsf{ind}_n + r \bmod L$ and then computes $k = t \bmod n$. P_2 provides P_0 and P_1 with shares $\langle E \rangle_0^L$ and $\langle E \rangle_1^L$ that reconstruct to E_k. Now, observe that $k = ((\mathsf{ind}_n + r) \bmod L) \bmod n$. Let $g = r \bmod n$. Since $n \mid L$, we have that $k = (\mathsf{ind}_n + g) \bmod n$. Now, let shares $\langle E \rangle_j^L = (\langle E^0 \rangle_j^L, \langle E^1 \rangle_j^L, \cdots, \langle E^{n-1} \rangle_j^L)$. In this, $\langle E^k \rangle_0^L$ and $\langle E^k \rangle_1^L$ reconstruct to 1, while all other $k' \neq k$ reconstruct to 0. Since $\langle D \rangle_j^L = (\langle E^{(-g \bmod n)} \rangle_j^L, \langle E^{(1-g \bmod n)} \rangle_j^L, \ldots, \langle E^{(n-1-g \bmod n)} \rangle_j^L)$, $\langle D^{(k-g) \bmod n} \rangle_0^L$ and $\langle D^{(k-g) \bmod n} \rangle_1^L$ alone will reconstruct to 1 with all other indices reconstructing to 0. Since $(k - g) \bmod n = \mathsf{ind}_n \bmod n$, we have that $\langle D \rangle_0^L$ and $\langle D \rangle_1^L$ reconstruct to E_{k^*}, hence proving the statement.

To argue security, first observe that P_0 and P_1 obtain shares of ind_n from the call to $\mathcal{F}_{\text{MAXPOOL}}$. Now, since r is uniformly random in $\mathbb{Z}_L$, P_2 learns no information from shares $\langle k \rangle_0^L$ and $\langle k \rangle_1^L$ (which reconstruct to $\mathsf{ind}_n + r$). Finally, $P_j, j \in \{0, 1\}$ only learn fresh shares of the values $E_{(\mathsf{ind}_n + r) \bmod n}$ and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh shares of the final output in Step 5 as the shares are randomized by U_0 and U_1. □

Derivative of Maxpool in the general case. We first observe that this function can be computed using steps similar to 6 & 7 from Algorithm 9. The idea is for the parties to invoke $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$ sequentially with shares of the unit vector representing the current maximum. Let E_k, $k \in [n]$ denote the unit vector of length n with 1 in its k^{th} position and 0 everywhere else. E_0 denotes the all zeroes vector. Details are presented in Algorithm 12.

Lemma 11. *Protocol* $\Pi_{\mathsf{DMP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 12 securely realizes* $\mathcal{F}_{\text{DMAXPOOL}}$ *in the* $(\mathcal{F}_{\text{DReLU}}, \mathcal{F}_{\text{SS}})-$*hybrid model.*

Proof. For correctness, we show that $\mathsf{Reconst}^L(\langle \mathsf{DMP}_n \rangle_0^L + U_0, \langle \mathsf{DMP}_n \rangle_1^L + U_1) = E_{k^*}$ in Algorithm 12. This proof is nearly identical to the proof of correctness of Algorithm 9. As before, we prove this by induction and

18 When we instantiate the functionality $\mathcal{F}_{\text{DReLU}}$ using protocol Π_{DReLU}, we would ensure that the conditions of Lemma 4 are met.

Algorithm 12 Derivative of Maxpool $\Pi_{\mathsf{DMP}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i \rangle_1^L\}_{i\in[n]}$, resp.

Output: P_0, P_1 get $\{\langle z_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle z_i \rangle_1^L\}_{i\in[n]}$, resp., where $z_i = 1$, when $x_i = \mathsf{Max}(\{x_i\}_{i\in[n]})$ and 0 otherwise.

Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L^n$ denoted by U_0 and U_1.

1: For $j \in \{0,1\}$, P_j sets $\langle \mathsf{max}_1 \rangle_j^L = \langle x_1 \rangle_j^L$ and $\langle \mathsf{DMP}_1 \rangle_j^L = E_j$.
2: **for** $i = \{2, \ldots, n\}$ **do**
3: For $j \in \{0,1\}$, P_j computes $\langle w_i \rangle_j^L = \langle x_i \rangle_j^L - \langle \mathsf{max}_{i-1} \rangle_j^L$
4: P_0, P_1, P_2 call $\mathcal{F}_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle w_i \rangle_j^L$ and P_0, P_1 learn $\langle \beta_i \rangle_0^L$ and $\langle \beta_i \rangle_1^L$, resp.
5: P_0, P_1, P_2 call $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i \rangle_j^L, \langle \mathsf{max}_{i-1} \rangle_j^L, \langle x_i \rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{max}_i \rangle_0^L$ and $\langle \mathsf{max}_i \rangle_1^L$, resp.
6: For $j \in \{0,1\}$, P_j sets $\langle K_i \rangle_j^L = E_{j \cdot i}$.
7: P_0, P_1, P_2 call $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i \rangle_j^L, \langle \mathsf{DMP}_{i-1} \rangle_j^L, \langle K_i \rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{DMP}_i \rangle_0^L$ and $\langle \mathsf{DMP}_i \rangle_1^L$, resp.
8: **end for**
9: For $j \in \{0,1\}$, P_j outputs $\langle \mathsf{DMP}_n \rangle_j^L + U_j$.

will maintain the invariant that max_i holds the value of $\mathsf{max}(x_1, \ldots, x_i)$ and now show that DMP_i holds the value E_k for k s.t. $\mathsf{max}_i = x_k$. It is easy to see that this holds for $i = 1$. Suppose this holds for $i = m - 1$. Then we will prove that it holds for $i = m$. Now, in Step 3, we calculate $w_m = x_m - \mathsf{max}_{m-1}$. By correctness of $\mathcal{F}_{\text{DReLU}}$, $\beta_m = \mathsf{ReLU}'(w_m)$. That is, $\beta_m = 1$ iff $x_m > \mathsf{max}_{m-1}$. Next, by correctness of $\mathcal{F}_{\text{SS}}$, max_m is max_{m-1} if $\beta_m = 0$ and x_m otherwise. In Step 6, we compute shares of $k_m = E_m$. In Step 7, by correctness of $\mathcal{F}_{\text{SS}}$, $\mathsf{DMP}_m = \mathsf{DMP}_{m-1}$ if $\beta_m = 0$ and E_m otherwise. This proves correctness.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$ provides outputs only to P_0 and P_1). Now, $P_j, j \in \{0,1\}$ only learn fresh shares of the values $\beta_i, \mathsf{max}_i, \mathsf{DMP}_i$ and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh shares of the final output in Step 9 as the shares are randomized by U_0 and U_1.

□