SecureNN: Efficient and Private Neural Network Training

Sameer Wagh[*] Divya Gupta[†] Nishanth Chandran[‡]

Abstract

Neural Networks (NN) provide a powerful method for machine learning training and prediction. For effective training, it is often desirable for multiple parties to combine their data – however, doing so conflicts with data privacy. In this work, we provide novel three-party and four-party secure computation protocols for various NN building blocks such as matrix multiplication, Rectified Linear Units, MaxPool, normalization etc. This enables us to construct three-party and four-party *information-theoretically secure* protocols for training and prediction of CNNs, DNNs and a number of other NN architectures such that no single party learns any information about the data.

Experimentally, we build a system and train a (A) 3-layer DNN (B) 4-layer CNN from MiniONN, and (C) 4-layer LeNet network. Compared to the state-of-the-art prior work SecureML (Mohassel and Zhang, IEEE S&P 2017) that provided (computationally-secure) protocols for only the network A in the 2 and 3-party setting, we obtain 93X and 8X improvements, respectively. In the WAN setting, these improvements are more drastic - for example, we obtain an improvement of 407X. Our efficiency gains come from a > 8X improvement in communication, coupled with the complete elimination of expensive oblivious transfer protocols. In fact, our results show that the overhead of executing secure training protocols is only between 17-33X of the cleartext implementation even for networks that achieve $> 99\%$ accuracy.

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 gets better as the amount of training data increases [40]. 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 compliance requirements [15, 6] or proprietary reasons. To enable training of NN models with good accuracy, it is highly desirable to be able 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 parties who own training data 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 $N = 3$ or 4, while M can be arbitrary.

[*]Princeton University; Email: snwagh@gmail.com. Work done in part while visiting Microsoft Research, India.
[†]Microsoft Research, India; Email: t-digu@microsoft.com
[‡]Microsoft Research, India, Email: nichandr@microsot.com

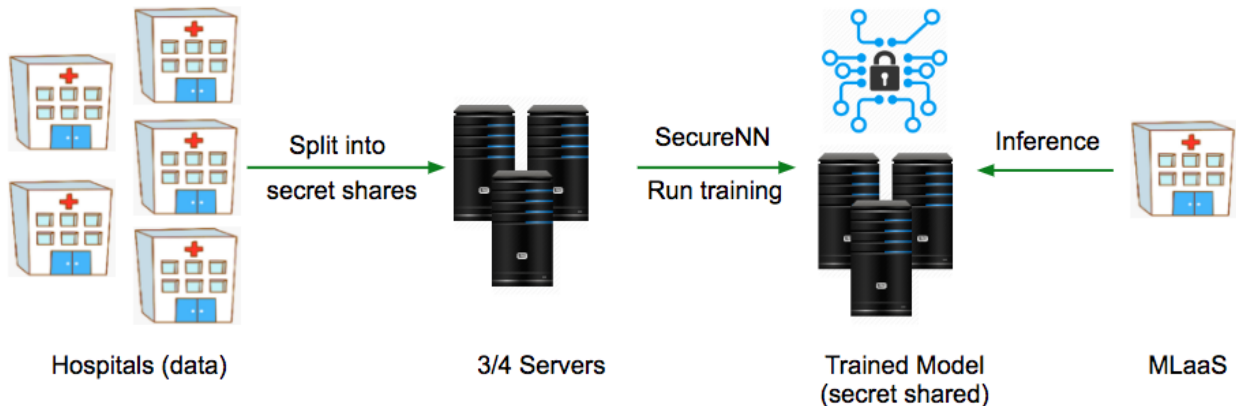


Figure 1: **Architecture**

The trained model is also kept hidden from any single server/party (and retained as secret shares). These secret shares can be put together by the servers (or by any other party) to reconstruct the model in the clear, if needed. If the model is retained as secret shares between the N servers, then inference (or prediction) can be executed with this trained model on any new input, while once again keeping the model, the new input point, and the output prediction, private from other parties as well as 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 that can be used as a Machine Learning as a Service (MLaaS) to help predict some disease or irregular health behaviour. This architecture is shown in Figure 1.

Fortunately, cryptography and specifically secure multi-party computation (MPC) [39, 24, 8, 18] provides a framework to solve the above problem. However, using general purpose MPC for complex tasks such as neural network training leads to highly inefficient, sometimes even "impossible-to-execute" protocols.

In this work, we build specialized protocols in the 3-server and 4-server settings that are approximately 8X–407X faster than, the state-of-the-art prior work, SecureML [31]. Our central result is the demonstration that the *price of secure NN training* over sensitive data can be quite low. For instance, the overhead of training a wide-class of NNs on a batch size of 128 using our 3-server protocols is only 17-33X that of training the same neural network over *cleartext data*. Comparatively, the state-of-the-art prior work SecureML [31] provided protocols for secure training in the 2-server and 3-server models for the same network and those protocols had an execution time of approximately 1634X and 140X that of our cleartext implementation. Additionally, in the 4-server model, the price of security can be brought down further and we achieve even lower overheads – only 13-22X compared to the cleartext cost. All our protocols enjoy information-theoretic security and are secure in the semi-honest adversarial model. (i.e., the adversary is not restricted to be a probabilistic polynomial time (PPT) algorithm, and tries to learn information through a faithful protocol execution). We are the first to demonstrate practicality to a broader class of NN training algorithms (CNNs). Our techniques are powerful enough to train neural networks that produce an inference accuracy of greater than 99% on the MNIST dataset [3] with practical overheads. Similarly, compared to recent works that considered only the problem of *secure inference*, we show that the overall execution time of our protocols are 42.4X faster than MiniONN [30], and 27X, 3.68X faster than concurrent and independent works of Chameleon [35] and Gazelle [27] respectively.

1.1 Our Contributions

Our main technical contribution is the construction of efficient 3-party (and, 4-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. We demonstrate the performance benefits of our protocols by running the tasks of secure training and secure inference on various neural networks, and comparing with the state of the art in privacy preserving machine learning as well as cleartext training.

Our Protocols. Computing neural networks requires repeated computing of a mix of matrix multiplications followed by non-linear activation functions such as ReLU and Maxpool. Hence, to be efficient we give protocols for evaluating all these functions in such a way that they can be combined easily. All our protocols maintain the invariant that parties begin with shares of input value(s) over a ring ($\mathbb{Z}_{2^{64}}$ in our case for system efficiency) and end the protocol with shares of the output value (again over the same ring). For example, if they begin with shares of matrices X and Y and execute matrix multiplication, they end the protocol with shares of Z, where $Z = X \cdot Y$.

We provide information-theoretic security against 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 inputs of the honest clients even if it runs in unbounded time[1]. When $N = 3$ or 4, as considered in our work, this is the best corruption threshold that one could hope to achieve against computationally unbounded adversaries [19]. We prove simulation-based security of in the Universal Composability (UC) framework [12, 13]. This means that the sub-protocols can be pieced together by the 3 (or 4) servers in an arbitrary manner to implement a wide-class of neural networks. Our main performance benefits come from avoiding the use of expensive garbled circuits to compute the non-linear activation functions. Our protocols have $> 8X$ improvement in communication complexity over garbled circuits based approach (that incurs security parameter multiplicative communication overhead to the data size) used in all previous works for computing non-linear functions. We provide a detailed overview of our techniques in Section 1.2.

Secure Training. To illustrate the generality and performance of our protocols, we implement training on 3 neural networks – (A) a DNN with three layers from SecureML [31], (B) A CNN from MiniONN [30] and (C) A 4-layer LeNet network [28]. We train all the networks on the MNIST data-set [3]. The *overall execution time* of our MPC protocol for Network A in the 3-server model over a LAN network is under an hour (roughly 52.8 minutes), while cleartext evaluation takes around 3 minutes on a similar machine. This illustrates that the cost of secure computation for neural network training can be within 20X of the no-security baseline. For our largest CNN network (Network C) with 99.15% accuracy, our protocols in the 3-server case executes in roughly 42.51 hours, while the corresponding cleartext execution time is 2.11 hours. We show how to modify our protocols in the 4-party setting to provide even better performance. For example, our 3-layer network has an execution time of 46.4 minutes and the 4-layer LeNet network has an execution time of 27.5 hours in the LAN setting.
Prior Work. The only prior work to consider secure training of neural networks is SecureML [31] that provides *computationally secure* protocols in the 2-server and 3-server models for Network A above (accuracy 93.4%). To train, their protocol in the 2-server model SecureML had an overall execution time of roughly 4900 minutes in the LAN setting and 4333 hours in the WAN setting, while their protocol in the 3-server model had an overall execution time of roughly 421.4 minutes

[1]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.

in the LAN setting. By constructing information-theoretic protocols that have an overall execution time of roughly 52.8 minutes in the LAN setting and 10.64 hours in the WAN setting, we illustrate improvements of 93X and 407X respectively over their 2-server results and 8X improvement in the 3-server model. Additionally, we show how to implement a much larger class of networks (including convolutional neural networks) that can obtain accuracies of $> 99\%$ on the MNIST dataset. [31] also split their protocols into an offline (independent of data) and online phase. Even when comparing only their online times with our *total times*, we obtain a 1.34X improvement in the LAN setting and a 5.6X improvement in the WAN setting over their 2-server results and a 3.26X improvement over their 3-server results[2]. Our drastic improvements can be attributed to 8X improvement in communication complexity and elimination of expensive oblivious transfer protocols, 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. There has been a significant effort on this problem and a series of recent works have made significant progress. The work of MiniONN [30] further optimizes the protocols of SecureML [31] (specifically for offline compute of matrix multiplications and convolutions) in the 2-server model. Concurrently and independently, the works of Chameleon [35] and Gazelle [27] also consider the problem of secure inference in the 3-server and 2-server models, respectively. Chameleon removes expensive oblivious transfer protocols (needed for secure multiplications) by using third party as dealer. Gazelle focused on making the linear layers (such as matrix multiplication and convolution) more communication efficient by providing specialized packing schemes for additively homomorphic encryption schemes. All of the previous schemes [31, 30, 35, 27] use expensive garbled circuits for activations. Our focus is on providing better protocols for the non-linear activation functions to reduce their communication complexity. As our experiments show, the *overall execution time* of our protocols are 42.4X faster than MiniONN, 27X faster than Chameleon and 3.68X faster than Gazelle. These gains come from a corresponding 38X, 1.63X and 4.05X reduction in communication over MiniONN, Chameleon, and Gazelle, respectively.

1.2 Techniques

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. While the cost of securely evaluating arithmetic layers can be high (and a lot of work has gone into reducing this cost [31, 30, 35, 27]), 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 (approximately 8X better) than the cost of converting to a Yao encoding and executing a garbled circuit.

Since we are in the 3-server setting, another tempting option might be to use techniques from Sharemind [9] and its successors [10, 14] that use techniques of bit extraction to perform the

[2] Due to the structure of their protocol, their 3-server protocol has a larger online time than their 2-server protocol while reducing the offline cost

Boolean computation. However, as shown in Chameleon [35], in the context of machine learning applications, the performance of these protocols is slower than the cost of converting to Yao's garbled circuit and executing a garbled circuits protocol. Furthermore, we show that by carefully constructing comparison and other Boolean protocols that avoid expensive bit extraction protocols, we can obtain much better performance than using garbled circuits. We now present our techniques in more detail.

In the 3-server setting, we denote the servers (or, parties) by P_0, P_1 and P_2. All our protocols maintain the invariant that P_0 and P_1 hold 2-out-of-2 shares of all intermediary computation values. We describe all our protocols in the three-party setting and point out differences with the 4-party version as appropriate.

Matrix multiplication and Convolutions. First, implementing information-theoretic matrix multiplication over shares when 3 parties are involved is straight-forward using matrix-based Beaver multiplication triplets [7] and is omitted from the discussion here. Convolutions are implemented in a very similar manner to matrix multiplication. In the four-party case, matrix multiplication is even easier. X_b for $b \in \{0, 1\}$ denote additive shares (over $\mathbb{Z}_L$) of a variable X. We have P_0 hold X_0, Y_0, P_1 hold X_0, Y_1, P_2 hold X_0, Y_1, and P_3 hold X_1, Y_0. Now, each party can now compute $X_i Y_j$ (for appropriate i, j) locally and once these shares are re-randomized (using shares of the zero matrix), we can ensure that P_0 and P_1 hold fresh shares of XY, without any party learning any information.

Non-linear activations. Our biggest challenge is in computing the non-linear activation functions such as $\mathsf{ReLU}'(x)$ (defined to be 1 if $x > 0$ and 0 otherwise) and $\mathsf{ReLU}(x)$ (defined to be $\mathsf{max}(x, 0)$). We compute these functions securely through a series of novel steps. 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 some field $\mathbb{Z}_p$) as well as a common random $\ell-$bit string r and a random bit β as input. This functionality computes the bit $(x > r)$ (which is 1 if $x > r$ and 0 otherwise) and XOR masks it with the bit β. This output is given to the third party P_2. We implement this functionality by building on the techniques of [20, 34] (who provided a computationally-secure protocol for a similar functionality) and provide an information-theoretic secure variant.

While it may seem that such a comparison protocol should suffice to implement $\mathsf{ReLU}'(x)$, unfortunately several barriers must be overcome. First, the above protocol required bits of the ℓ-bit x to be secret shared between P_0 and P_1 over a field $\mathbb{Z}_p$. While we could potentially execute our entire protocol over this field $\mathbb{Z}_p$, this would lead to the following severe inefficiencies. To begin, the comparison protocol required *bits of* x to be shared between P_0 and P_1. However, secure matrix multiplication is highly inefficient if we were to work with Boolean values (which we would have to if parties start out with shares of bits of x). To overcome this, we define a 3-party functionality (and provide a corresponding protocol), that we call $\mathcal{F}_{\text{MSB}}$, that allows x to be secret shared as an element over a ring $\mathbb{Z}_N$ and still computes the $\mathsf{MSB}(x)$. This protocol exploits the fact that computing the MSB of a value x over a ring $\mathbb{Z}_N$ is equivalent to computing the LSB of $2x$ over the same ring as long as N is odd. It then makes use of our previous comparison protocol to perform the comparison *without actually having to convert* x *from its ring representation to bits*, which is where we get high efficiency gains.

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 [31]), 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. We then 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 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 all these protocols, we obtain our protocol for computing $\mathsf{ReLU}'(x)$ (the derivative of ReLU). Having constructed a protocol that computes $\mathsf{ReLU}'(x)$, we use this along with the multiplication protocol to compute $\mathsf{ReLU}(x)$.

Division, normalization and Maxpool is implemented using the $\mathsf{ReLU}'(\cdot)$ and multiplication protocols, while an efficient protocol for the derivative of maxpool is constructed exploiting specific number-theoretic properties.

Putting it all together. Finally, we demonstrate how to tie these various sub-protocols to obtain protocols for training various neural network. Since we prove the simulation security of all our protocols in the universal composability framework, combining and proving the security of our final protocols is easy.

1.3 Organisation of the paper

We describe the security model and the neural network training algorithms that we use in Section 2. Section 3 contains our low level 3-server protocols that are used as building blocks in our main protocols for various functionalities. We point out our modifications to the 4-server case here as well. In Section 4, 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 5. We present the detailed evaluation of our experiments in Section 6. We present related work in Section 7 and conclude in Section 8. Due to lack of space, we defer further details of security, machine learning algorithms, ideal functionalities, security proofs and further details of our 4-party protocols to the Appendix.

2 PRELIMINARIES

2.1 Threat Model and Security

We will model and prove the security of our construction in the simulation paradigm [25, 12, 13]. At a very high level, 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 machine $\mathcal{F}$ that carries 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. In our setting, the adversary corrupts at most one party and learns everything that this party sees during the protocol. The adversary is *semi-honest*, and follows the protocol specification honestly (also known as honest-but-curious security). The adversary is not restricted to run in polynomial time (i.e., we provide information-theoretic security), but we require honest parties to be polynomial time machines. As is standard in all information-theoretic protocols, we only assume point-to-point secure channels between all pairs of parties in the protocol. Finally, protocols typically invoke other sub-protocols. In this framework, the *hybrid model* is like a real interaction, except that some invocations of the sub-protocols are replaced by the invocation of an instance of an ideal functionality $\mathcal{F}$; this is called the "$\mathcal{F}$-hybrid model". For further details on our threat model and security, we refer the reader to Appendix B.1.

2.2 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 backward propagation 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 is used for minimizing the loss.

A large class of networks can be represented using the following functions: matrix multiplication, convolution, ReLU, MaxPool (which is defined as the maximum of a set of values, usually in a sub-matrix), normalization (which is defined to be $\frac{x_i}{\sum x_i}$ 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 [31]) that provides an inference accuracy of 93.4% on the MNIST data set [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 one more neural network from Chameleon [35]. More details on these networks are presented in Appendix B.2.

3 Protocols

In this section, we will mostly focus on the 3-party protocols and point out the main optimizations that can be done in the 4-party setting. We will describe all our 3-party protocols and prove their correctness and semi-honest simulation based security against a single corruption. We do this in three steps: first, in Section 3.1, we provide protocols for "supporting functionalities" (see Appendix D); these are funtionalities and protocols that will be used as building blocks to construct protocols for our main functionalities. in Section 3.2, we describe the modifications to supporting protocols in the four party case. Next, in Section 4, we provide protocols for our "main functionalities"; these are functionalities that correspond to various neural network functions and layers such as linear layer, convolution layer, ReLU, and so on. Finally, in Section 4.7, we outline how to put these main protocols together to obtain protocols for a large class of neural networks. We now set notation.

Notation. In our protocols, we use additive secret sharing over the four rings $\mathbb{Z}_L$, $\mathbb{Z}_{L-1}$, $\mathbb{Z}_p$ and $\mathbb{Z}_2$, 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. 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 also use $\langle x \rangle^t$ to denote sharing of x over $\mathbb{Z}_t$ (we abuse notation and write $\langle x \rangle^B$ to denote sharing of x over $\mathbb{Z}_2$). The algorithm $\mathsf{Share}^t(x)$ generates the two shares of x over the ring $\mathbb{Z}_t$ and algorithm $\mathsf{Reconst}^t(x_0, x_1)$ reconstructs a value x using x_0 and x_1 as the two shares over $\mathbb{Z}_t$ (reconstruction is simply $x_0 + x_1$ over $\mathbb{Z}_t$). Also, for any $\ell-$bit integer x, we use $x[i]$ to denote the ith bit of x. Then, $\{\langle x[i] \rangle^t\}_{i \in [\ell]}$ denotes the shares of bits of x over $\mathbb{Z}_t$. For an $m \times n$ matrix X, we denote by $\langle X \rangle_0^t$ and $\langle X \rangle_1^t$ the matrices that are created by secret sharing the elements of X component-wise (other notation on X, such as $\mathsf{Reconst}^t(X_0, X_1)$ is similarly defined component-wise).

We assume that parties P_0, P_1 preshare fresh shares of 0 (that can be generated using common

randomness between pair of parties). These shares of 0 can be used to refresh the secret shares by each party locally adding its share of 0 to the share that needs to be refreshed. 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. In the following, 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 4), 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 4.7) 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 output.

3.1 Supporting Protocols

In this section, we describe various building blocks to our main protocols. These protocols sometime 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 sometimes receives output in these protocols.

3.1.1 Matrix Multiplication

Algorithm 1 describes our 3-party protocol for secure multiplication (functionality $\mathcal{F}_{\text{MATMUL}}$ in Figure 2, Appendix D) 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 triplets [7] generalized to the matrix setting. P_2 generates these triplet shares and sends to parties P_0, P_1. The proof of the following lemma is provided in Appendix E.

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 $-jE \cdot F + \langle X \rangle_j^L \cdot F + E \cdot \langle Y \rangle_j^L + \langle C \rangle_j^L + U_j$.

Lemma 1. *Protocol* $\Pi_{\mathsf{MalMul}}(\{P_0, P_1\}, P_2)$ *in Algorithm 1 securely realizes* $\mathcal{F}_{\text{MATMUL}}$ *(see Figure 2, Appendix D).*

3.1.2 Select Share

Algorithm 2 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{SS}}$ (see Figure 11, Appendix D). Parties P_0, P_1 hold shares of x, y over $\mathbb{Z}_L$ and also of a selection bit $\alpha \in \{0, 1\}$ over $\mathbb{Z}_L$.

Parties P_0, P_1 get fresh shares of x if $\alpha = 0$ and fresh shares of y if $\alpha = 1$.
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 $\mathcal{F}_{\text{MATMUL}}$ for multiplying α and $(y - x)$.

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 call $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ 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 $\langle z\rangle_j^L = \langle x\rangle_j^L + \langle c\rangle_j^L + u_j$.

Lemma 2. *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.*

3.1.3 Private Compare

Algorithm 3 describes our three-party protocol realizing the functionality $\mathcal{F}_{\text{PC}}$ (Figure 8, Appendix D). The parties P_0 and P_1 holds shares of bits of x in $\mathbb{Z}_p$, 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 $\beta' = \beta \oplus (x > r)$.
Intuition. Our starting point is the idea for 2-party comparison present in [20, 34]. We build on this to give a much more efficient information theoretic 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 have a corner case of $r = 2^\ell - 1$ that we discuss 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 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 $x > r$. (For formal proof, see Appendix E.) Hence, both P_0, P_1 send shares of c_i to P_2 who reconstructs c_i by adding and looks for 0. To ensure security against a corrupt P_2, we hide exact values of non-zero c_i and index of 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 $\mathbb{Z}_p$. We prove the correctness and security of the lemma below formally in Appendix E.

Lemma 3. *Protocol $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ in Algorithm 3 securely realizes $\mathcal{F}_{\text{PC}}$ (Figure 8, Appendix D) when $p > \ell + 2$.*

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'.

3.1.4 Share Convert

Algorithm 4 describes our three-party protocol for converting shares over $\mathbb{Z}_L$ to $\mathbb{Z}_{L-1}$ realizing the functionality $\mathcal{F}_{\mathrm{SC}}$ (see Figure 9, Appendix D). 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, we use $\kappa = \mathsf{wrap}(x, y, L)$ to denote $\kappa = 1$ 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.

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 call $\mathcal{F}_{\mathrm{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $\left(\{\langle x[i]\rangle_j^p\}_{i\in[\ell]}, r, \eta''\right)$ 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 $L-1$)

Lemma 4. *Protocol $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ in Algorithm 4 securely realizes $\mathcal{F}_{\mathrm{SC}}$ (Figure 9, Appendix D) in the $\mathcal{F}_{\mathrm{PC}}$−hybrid (Figure 8, Appendix D) model.*

Proof. Here, we prove the correctness of our protocol and defer the proof of security to Appendix E. 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 functionality $\mathcal{F}_{\mathrm{PC}}$, $\eta' = \eta'' \oplus (x > r)$. Next, let $\eta = \mathsf{Reconst}^{L-1}(\langle \eta\rangle_0^{L-1}, \langle \eta\rangle_1^{L-1}) = \eta' \oplus \eta'' = (x > r)$. Next, note that $x \equiv a + r \mod L$. Hence, $\mathsf{wrap}(a, r, L) = 0$ iff $x > r$. 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. □

3.1.5 Compute MSB

Algorithm 5 describes our three party protocol realizing the functionality $\mathcal{F}_{\text{MSB}}$ (see Figure 10, Appendix D) that computes the most significant bit (MSB) of a value $a \in \mathbb{Z}_{L-1}$. Parties P_0, P_1 hold shares of a over odd ring $\mathbb{Z}_{L-1}$ and end up with secret shares of $\mathsf{MSB}(a)$ over $\mathbb{Z}_L$.

Intuition: Borrowing from [34], note that when the shares of the private input (say a) are over an odd ring (such as after using $\mathcal{F}_{\text{SC}}$), the MSB computation can be converted into a 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 a call to private compare functionality $\mathcal{F}_{\text{PC}}$. 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).).

Lemma 5. *Protocol $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ in Algorithm 5 securely realizes $\mathcal{F}_{\text{MSB}}$ (Figure 10, Appendix D) in the $(\mathcal{F}_{\text{PC}}, \mathcal{F}_{\text{MATMUL}})$-hybrid (Figure 8, Appendix D) model.*

Correctness follows along the intuition provided above and we give formal proofs of correctness and security in Appendix E.

3.2 Modifications in the 4-party case

In the 4-party case, the main change is in the way matrix multiplication is implemented. We do not require Beaver triplets and can do matrix multiplication much more efficiently in the following way. For the secure multiplication of matrices X and Y, the parties P_0 and P_1 hold shares of X and Y, where $X \in \mathbb{Z}_L^{m \times n}$ and $Y \in \mathbb{Z}_L^{n \times v}$; parties P_2 and P_3 have no input. At the end of the protocol, P_0 and P_1 learn shares of matrix $X \cdot Y$. If $\langle X \rangle_j^L, j \in \{0, 1\}$ are the shares of X and $\langle Y \rangle_j^L, j \in \{0, 1\}$ are the shares of Y, P_0 sends $\langle X \rangle_0^L$ to P_2 and $\langle Y \rangle_0^L$ to P_3; similarly, P_1 sends $\langle X \rangle_1^L$ to P_2 and $\langle Y \rangle_1^L$ to P_3. Now, every party computes the $\langle X \rangle_i^L j$ value that they can (and appropriately randomize these shares). Now, note that the sum of all shares held by all 4 parties is indeed $X \cdot Y$. Hence, P_2 and P_3 can send their respective shares to P_0 and P_1 (after re-randomizing) to complete the protocol. The protocol is described in Algorithm 6. This protocol has a communication

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 $\mathcal{F}_{\text{PC}}(\{P_0, P_1\}, P_2)$ 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 $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1, P_2\})$ 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 $\langle \alpha \rangle_j^L = \langle \gamma \rangle_j^L + \langle \delta \rangle_j^L - 2\langle \theta \rangle_j^L + u_j$.

complexity that is 1.66X smaller than the corresponding 3-party protocols resulting in more efficient protocols. The proof of the following lemma is easy to see and is provided in the Appendix E.

Algorithm 6 Mat. Mul. $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2, P_3)$:

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 as well as P_2 and P_3 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 $\langle 0^{m \times v} \rangle_1^L = U_1$, P_2 holds $\langle 0^{m \times v} \rangle_0^L = V_0$, and P_3 holds $\langle 0^{m \times v} \rangle_1^L = V_1$

1: P_0 sends $\langle X \rangle_0^L$ to P_2 and $\langle Y \rangle_0^L$ to P_3.
2: P_1 sends $\langle Y \rangle_1^L$ to P_2 and $\langle X \rangle_1^L$ to P_3.
3: P_2 computes $\langle W \rangle_0^L = \langle X \rangle_0^L \cdot \langle Y \rangle_1^L + V_0$ and P_3 computes $\langle W \rangle_1^L = \langle X \rangle_1^L \cdot \langle Y \rangle_0^L + V_1$.
4: P_2 sends $\langle W \rangle_0^L$ to P_0 and P_3 sends $\langle W \rangle_1^L$ to P_1.
5: $P_j, j \in \{0,1\}$ outputs $\langle Z \rangle_j^L = \langle X \rangle_j^L \cdot \langle Y \rangle_j^L + \langle W \rangle_j^L + U_j$.

Lemma 6. *Protocol* $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2, P_3)$ *in Algorithm 6 securely realizes* $\mathcal{F}_{\text{MATMUL}}$.

We make minor modifications to some other protocols in the 4-party setting as well, which we present in the full version of this paper.

Protocol	3PC		4PC	
	Rounds	Comm.	Rounds	Comm.
$\mathsf{MatMul}_{m,n,v}$	2	$2(2mn+2nv$ $+mv)\ell$	2	$2(mn+nv$ $+mv)\ell$
SS	2	10ℓ	2	$6\ell+4$
PC	1	$2\ell\log p$	1	$2\ell\log p$
SC	4	$4\ell\log p+6\ell$	4	$4\ell\log p+6\ell$
MSB	5	$4\ell\log p+16\ell+2$	4	$4\ell\log p+2\ell+4$

Table 1: Round & comm. complexity of sub-protocols.

3.3 Overheads of supporting protocols

The communication and round complexity of our supporting protocols for 3PC and 4PC 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. All other complexities are provided for single elements.

4 Main Protocols

In this section, we describe all our main protocols for functionalities such as linear layer, derivate of ReLU, ReLU and so on. 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. Party P_2 takes the role of "assistant" in all protocols and has no input.

4.1 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. [5] for an exposition on convolutional layers. Hence both these layers can be directly implemented using Algorithm 1 from Section 3.1.

4.2 Derivative of ReLU

Algorithm 7 describes our three party protocol for realizing the functionality $\mathcal{F}_{\text{DRELU}}$ (see Figure 3, Appendix D) that computes the derivative of Relu, denoted by ReLU', at 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 $\mathsf{ReLU}'(a) = 1$ if $\mathsf{MSB}(a) = 0$, else $\mathsf{ReLU}'(a) = 0$.

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 functionality $\mathcal{F}_{\text{MSB}}$ 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 $\mathcal{F}_{\text{SC}}$ we require that value is not equal to $L-1$. This is ensured by first computing shares of $c = 2a$ and then calling $\mathcal{F}_{\text{SC}}$. We ensure[3] that $\mathsf{ReLU}'(a) = \mathsf{ReLU}'(c)$ by requiring that $a \in [0, 2^k] \cup [2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$. We provide a formal proof of the security lemma below in Appendix E.

Algorithm 7 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 call $\mathcal{F}_{\text{SC}}(\{P_0, P_1\}, P_2)$ 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 call $\mathcal{F}_{\text{MSB}}(\{P_0, P_1\}, P_2)$ 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 $\langle \gamma \rangle_j^L = j - \langle \alpha \rangle_j^L + u_j$.

Lemma 7. *Protocol* $\Pi_{\mathsf{DRELU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 7 securely realizes* $\mathcal{F}_{\text{DRELU}}$ *(Figure 3, Appendix D) 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* $k < \ell - 1$.

4.3 ReLU

Algorithm 8 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\text{RELU}}$ (see Figure 4, Appendix D) 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, $\mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$.

Intuition: Our protocol implements the above relation by using one call each to $\mathcal{F}_{\text{DRELU}}$ and $\mathcal{F}_{\text{MATMUL}}$. Note that $\mathcal{F}_{\text{MATMUL}}$ is invoked for multiplying two matrices of dimension 1×1 (or just one integer multiplication). The security lemma that we show in the Appendix is stated below.

Algorithm 8 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 call $\mathcal{F}_{\text{DRELU}}(\{P_0, P_1\}, P_2)$ 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 $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$ 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 $\langle c \rangle_j^L + u_j$.

Lemma 8. *Protocol* $\Pi_{\mathsf{ReLU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 8 securely realizes* $\mathcal{F}_{\text{RELU}}$ *(Figure 4, Appendix D) in the* $(\mathcal{F}_{\text{DRELU}}, \mathcal{F}_{\text{MATMUL}})$*-hybrid model.*

[3] 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 all the ML applications that we work with.

4.4 Division

We discuss our 3-party protocol realizing the functionality $\mathcal{F}_{\text{DIV}}$ (see Figure 5, Appendix D). 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 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 $\mathcal{F}_{\text{DRELU}}$. 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 divident. This selection can be implemented using $\mathcal{F}_{\text{MATMUL}}$ (similar to ReLU computation). Hence, division protocol proceeds in iterations and each iteration makes one call to $\mathcal{F}_{\text{DRELU}}$ and two calls to $\mathcal{F}_{\text{MATMUL}}$. Due to space constraints, we provide the protocol in Algorithm 13, Appendix E.

4.5 Maxpool

Algorithm 9 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{MAXPOOL}}$ (see Figure 6, Appendix D) to compute 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 $\mathcal{F}_{\text{SS}}$ 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 back-propagation during training.

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.*

4.6 Derivative of Maxpool

The derivative of the maxpool function is defined as the unit vector with a 1 only in the position with the maximum value (see functionality $\mathcal{F}_{\text{DMAXPOOL}}$ presented in Figure 7 in Appendix D). 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, see Algorithm 14, Appendix E.

Intuition. The key observation behind this protocol is that when n divides L (i.e., $n|L$), we have that $a \bmod n = (a \bmod L) \bmod n$. The first step that P_0 and P_1 run is $\mathcal{F}_{\text{MAXPOOL}}$ 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 $rin\mathbb{Z}_n$ and have P_2 reconstruct $k = (\mathsf{ind} + r) \bmod n$ by sending shares of r and ind (over $\mathbb{Z}_L$) to P_2. 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 $n|L$. The proof of the lemma below is in the Appendix.

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 $\mathcal{F}_{\text{DRELU}}(\{P_0, P_1\}, P_2)$ 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 $\langle k_i\rangle_j^L = j \cdot i$.
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 $(\langle \beta_i\rangle_j^L, \langle \mathsf{ind}_{i-1}\rangle_j^L, \langle k_i\rangle_j^L)$ 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 $(\langle \mathsf{max}_n\rangle_j^L + u_j, \langle \mathsf{ind}_n\rangle_j^L + v_j)$.

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

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 $\langle D\rangle_j^L + U_j$.

Protocol	Rounds	Communication
$\mathsf{Linear}_{m,n,v}$	2	$2(2mn + 2nv + mv)\ell$
$\mathsf{Conv2d}_{m,i,f,o}$	2	$2(2m^2 f^2 o + 2f^2 oi + m^2 o)\ell$
DReLU	8	$8\ell \log p + 22\ell + 4$
ReLU	2	10ℓ
NORM(l_D) or DIV(l_D)	$10l_D$	$(8\ell \log p + 42\ell + 4)l_D$
$\mathsf{Maxpool}_n$	$9n$	$(8\ell \log p + 32\ell + 4)n$
DMP_n	2	$2(n+1)\ell$

Table 2: 3PC Main protocols: Comm. & Round Complexity

Lemma 10. $\Pi_{\mathsf{n_1xn_2DMP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 10 securely realizes* $\mathcal{F}_{\text{DMAXPOOL}}$ *in the* $\mathcal{F}_{\text{DMAXPOOL}}$*−hybrid model.*

4.7 Neural Network Protocols

Our main protocols can be put together in an easy manner to execute training on a wide class of neural networks. For example, consider the 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 Appendix B.2). To implement this, we first invoke $\mathcal{F}_{\text{MATMUL}}$, followed by $\mathcal{F}_{\text{RELU}}$, then again followed by $\mathcal{F}_{\text{MATMUL}}$ and $\mathcal{F}_{\text{RELU}}$ and finally we invoke $\mathcal{F}_{\text{DIV}}$ to compute $\mathsf{ASM}(\cdot)$[4]. Back propagation is computed by making calls to $\mathcal{F}_{\text{MATMUL}}$ as well and $\mathcal{F}_{\text{DRELU}}$ with appropriate dimensions. 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.

5 Communication and Rounds

The communication and round complexity of our protocols for the various ML functionalities are presented in Table 2 for the 3-party case and in Table 3 for the 4-party case. $\mathsf{Linear}_{m,n,v}$ 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).

Our gains mainly come from the secure evaluation of non-linear functions such as ReLU and $\mathsf{Maxpool}$ and their derivatives. SecureML [31] took a garbled circuit approach to evaluate these functions – i.e., after completion of an arithmetic (linear) computation such as matrix multiplication, they ran a protocol to convert shares of intermediary values into encoding suitable for garbled

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

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 encoding 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], as well as used in SecureML, this cost is $6\kappa\ell$, where κ is the security parameter). On the other hand, in our approach, we provide new protocols to compute such non-linear activation functions. For example, the ReLU protocol that we construct avoids paying κ multiplicative overhead and has communication complexity of $8\ell \log p + 32\ell + 4$, which is approximately 96ℓ (when $p = 67$ as is in our setting). This leads to an 8X improvement in the communication complexity of the protocols for non-linear functions. When computing linear functions, we use the same Beaver triplet technique as [7, 31, 35] in the 3-party case and provide more efficient protocols in the 4-party case with a 1.66X improvement in communication.

6 Evaluation

System Details. We test our prototype by running experiments over Amazon EC2 c4.8x large instances in two environments – one that models a LAN setting and another, the WAN setting. Our system is implemented in about 7200 lines of C++ code with the use of standard libraries. The ring size is set to $\mathbb{Z}_{2^{64}}$ and we use the `uint64_t` native C++ datatype for all variables. As noted in [31], compared to using a field for the underlying protocols or using dedicated number theoretic libraries such as NTL [4], this has the benefit of implementing modulo operations for free. We use the Eigen Library [1] for faster matrix multiplications (both in the MPC setting and the stand-alone, single party code which provides the benchmark when no security is involved).
LAN setting. We use 3 (and 4 respectively) Amazon EC2 c4.8xlarge machines running Ubuntu in the same region. At the time of running the experiments, the average bandwidth was 625MB/s and the average ping time was 0.215ms.
WAN setting. In the WAN setting, we rent machines in different regions (Oregon, Ohio and North California) with the same machine specifications as in the LAN setting. At the time of running the experiments, the average bandwidth was 40MB/s and the average ping time was 69ms.
Number encoding. Typically 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 integer/fixed point form. We use the methodology from [31] to support decimal arithmetic in an integer ring (described in Appendix C). In all our experiments we use $\ell = 64$ and 13 bits of precision (cleartext training is also done with these parameters).

Summary of experiments. We develop a prototype that fully implements end-to-end secure three and four-party neural network training algorithms. We test the performance of our protocols by evaluating over 3 different neural networks that train over the MNIST dataset [3]. We also demonstrate the performance of secure inference in Section 6.3. Finally, in Section 6.4, we also present microbenchmarks that measure the performance of various sub-protocols 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. We are also the first to compare the performance of our protocols to cleartext (insecure) training and make a strong case for the practicality of secure neural network training and inference[5].

[5]We compare our protocols with our stand-alone C++ implementation of the corresponding neural networks. We do not consider optimized ML implementations such as Tensorflow or Pytorch.

Protocol	Rounds	Communication
$\mathsf{Linear}_{m,n,v}$	2	$2(mn + nv + mv)\ell$
$\mathsf{Conv2d}_{m,i,f,o}$	2	$2(m^2 f^2 o + f^2 oi + m^2 o)\ell$
DReLU	7	$8\ell \log p + 8\ell + 4$
ReLU	2	$6\ell + 2$
$\mathrm{NORM}(l_D)$ or $\mathrm{DIV}(l_D)$	$9l_D$	$(8\ell \log p + 20\ell + 8)l_D$
$\mathsf{Maxpool}_n$	$9n$	$(8\ell \log p + 14\ell + 6)n$
DMP_n	2	$2(n+1)\ell$

Table 3: 4PC Main protocols: Comm. & Round Complexity

Our times for secure training are extrapolated from 10 iterations and for inference all times are averaged over 10 executions. We consider overall execution time (and do not split times into an offline, data independent phase, and an online, data dependent phase). The learning rate is 2^{-5} in all experiments, except in the SecureML [31] network, where we retain their learning rate of 2^{-7}.

6.1 Neural Networks

Benchmarks. We consider three networks that perform training over MNIST dataset for handwritten digit recognition. This dataset contains 60,000 training samples of handwritten digits. Each image is a 28*28 square image, with each pixel represented using 1 Byte. The inference set contains 10,000 images. We use these networks for training as well as inference.

- **Network A.** The first network is the Deep Neural Network from [31] which is a 3-layer network comprising of only fully connected (linear) layers and uses ReLU as the activation function. This network, after training for 15 epochs, has a prediction accuracy of 93.4% as illustrated in [31].
- **Network B.** The 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 network comprises of 2 convolutional layers followed by 2 linear layers and uses ReLU as well as Maxpool as its activation functions. We show that this network, after training for 15 epochs, has a prediction accuracy of 98.77%.
- **Network C.** Finally, we also run our protocols over the (standard) LeNet network [28], which is a larger version of the network from [30]. We show that this network, after training for 15 epochs, has a prediction accuracy of 99.15%.

In addition to these networks for training, for the case of secure inference, we also consider a network from Chameleon [35] for comparison in the 3-party setting. This network has one convolutional layer and two fully connected layers and uses ReLU as the activation function. This network gives inference accuracy of 99%. We will refer to this network as **Network D**. For a detailed description of these networks see Appendix B.2.

6.2 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. We remark that we are the first work to show the feasibility of secure training on large and complex NNs that achieve high levels of accuracy. We present times for cleartext execution, 3PC and 4PC. 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 4 presents a summary of our results in the LAN/WAN setting in comparison with cleartext execution (no security baseline) as a function of the number of epochs for training (batch size fixed to 128), while Table 5 presents the results when the batch size is varied and the number of epochs is fixed to 5.

	Epochs	Accuracy	Cleartext	3PC (hr)		4PC (hr)
				LAN	WAN	LAN
A	15	93.4%	0.05	0.88	10.64	0.78
B	5	97.94%	0.27	8.96	40.3	6.21
	10	98.05%	0.54	17.93	80.6	12.43
	15	98.77%	0.81	26.89	120.9	18.64
C	5	98.15%	0.7	14.17	77.38	9.18
	10	98.43%	1.41	28.34	154.77	18.36
	15	99.15%	2.11	42.51	232.15	27.54

Table 4: Secure training execution times for batch size 128.

	Batch size	Accuracy	Cleartext	3PC (hr)		4PC (hr)
				LAN	WAN	LAN
B	4	99.15%	0.35	14.65	149.58	10.19
	16	98.99%	0.23	11.74	66.61	7.99
	128	97.94%	0.27	8.96	40.3	6.21
C	4	99.01%	1.13	23.04	170	15.04
	16	99.1%	0.66	18.46	80.57	12.11
	128	98.15%	0.7	14.17	77.38	9.18

Table 5: Secure training execution times for 5 epochs.

	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	-	-	-
	Our 3PC	0	0.88	0.88	0	10.64	10.64

Table 6: Training time comparison for Network A for batch size 128 and 15 epochs with SecureML [31].

LAN setting. As can be seen from the timings, we obtain secure protocols that have an overhead of between 17-51X when comparing with an insecure (cleartext), stand-alone execution. In the four-party setting, this ratio is between 13-34X. **WAN Setting.** In this setting, we obtain secure protocols that have an overhead of between 110-427 times when comparing with the cleartext execution. In fact, all our protocols (even the more complex neural networks) have an execution time of only a few hundred hours. *Given that the WAN setting is a more natural and representative setup for MPC protocols relying on non-colluding parties, we feel that our work serves as a key enabler for such a technology.*

Comparison with prior work. The only prior work to consider neural network training was SecureML [31] (Network A). They considered both 2 and 3 party training on a 3-layer DNN. Our 3 party protocol is roughly 8X faster than their 3 party protocol and 93X faster than their 2 party protocol. Our 4 party protocol is faster the our 3 party protocol and hence we get even better improvements there. 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 1.33X over their 2PC and a 3.26X improvement over their 3PC (their 3PC trades off some offline cost with a larger online cost). In the WAN setting, our improvements are even more dramatic and we get an improvement of 407X. Refer to Table 6 for details.

6.3 Secure Inference

We also evaluate our protocols for the task of secure inference for the networks A, B, C and D. These networks can either be a result of secure training using 3PC or 4PC protocol and are secret shared between P_0 and P_1, or they can be secret shared between these parties 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 7 summarizes our comparison with state-of-the-art secure inference protocols. Networks A and B were considered in SecureML [31], MiniONN [30] and Gazelle [27] using different techniques for secure computation between 2 parties. Each of these works split their compute into an input independent offline phase and an input dependent online phase. In our protocols, we don't have any offline phase and hence, the offline cost is 0. Our protocols in the 3PC setting achieve at least 2X improvement in small networks that have a small number of non-linear operations (such as Network D) and between 4-38X improvements in larger networks. The corresponding numbers in the 4PC setting are 3X and between 5.2-49X. In fact, in all cases, our total time is lower than the online time of previous best protocols (ignoring the

	Framework	Runtime (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML	4.7	0.18	4.88	-	-	-
	Our 3PC	0	0.05	0.05	0	4.03	4.03
	Our 4PC	0	0.04	0.04	0	2.08	2.08
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
	Our 3PC	0	0.22	0.22	0	17.28	17.28
	Our 4PC	0	0.16	0.16	0	13.43	13.43
C	Our 3PC	0	0.34	0.34	0	37.03	37.03
	Our 4PC	0	0.23	0.23	0	25.49	25.49
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
	Our 3PC	0	0.1	0.1	0	7.93	7.93
	Our 4PC	0	0.067	0.067	0	5.33	5.33

Table 7: Single image inference time comparison of various protocols in the LAN setting.

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 for both 3PC and 4PC. Finally, for network D, we also compare our protocols with the 3PC protocols in Chameleon [35]. This benchmark shows that our 3PC beats the state-of-the-art for 3PC by 27X.

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 7). 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 8 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 $0.41s$ and $3.6s$ for 128 predictions using 3PC protocol.

6.4 Microbenchmarks

Table 9 presents microbenchmark timings for our various ML functionality protocols varied across different dimensions. All timings are average timings. While we have reduced the timings of non-linear function computations significantly, as can be seen from the table, they do have much

	Runtime 3PC (s)				Runtime 4PC (s)	
	LAN		WAN		LAN	
Batch size →	1	128	1	128	1	128
A	0.05	0.41	3.1	3.6	0.04	0.256
B	0.22	11.37	5.15	63.52	0.16	9.71
C	0.34	16.41	5.33	97.53	0.23	14.64
D	0.1	3.75	3.96	19.63	0.067	2.83

Table 8: Prediction timings for batch size 1 vs 128 for our protocols on Networks A-D over MNIST.

higher cost than computation of linear layers and convolutional layers.

7 Related Work

Most prior works have focused on privacy-preserving machine learning prediction. As discussed earlier, the only work to consider neural network training is SecureML [31]. The work of MiniONN [30] builds on [31] and constructs more efficient protocols for neural network *prediction*. The concurrent and independent works of Chameleon [35] and Gazelle [27] build protocols for 3 and 2 party neural network prediction. Chameleon primarily makes use of a third party to avoid expensive oblivious transfer protocols in SecureML, while Gazelle builds better packing mechanisms for additively homomorphic encryption schemes based on lattices that bring down the cost of linear layers such as matrix multiplication and convolution. Their protocols for non-linear activation functions such as ReLU and Maxpool remain the same as in SecureML. On the other hand, we provide more communication efficient protocols for non-linear activation functions (which form the largest overhead in secure neural network protocols) while implementing the linear layers using Beaver triplets for 3-party computation and a much more communication efficient protocol for 4-party computation. Other frameworks, such as Sharemind [9] provide a way to obtain generic 3-party secure computation. As discussed in Chameleon [35], the protocols obtained from these frameworks are at least an order of magnitude slower than [35] (whose performance itself we better). We provide a more detailed overview of other related works in Appendix A.

8 Conclusions

We develop new 3 and 4-party information-theoretically secure protocols for neural network training and prediction such that no single party learns any information about the data. Our key contribution is more communication efficient protocols for non-linear activation functions. Experimentally, we test our protocol over a wide class of networks and demonstrate the practicality of our approach. We obtain *several orders of magnitude improvements* over previous state-of-the-art protocols (that trained smaller, lower accuracy networks). Our results indicate that the overhead of executing the secure MPC protocols is between 13-33X of the cleartext implementation even for larger networks achieving $> 99\%$ accuracy over the MNIST dataset.

Protocol	Dimension	Runtime (ms)		Comm. (MB)	
		3PC	4PC	3PC	4PC
$\textsf{Conv2d}_{m,f,i,o}$	$8, 5, 16, 50$	39.3	18.2	8.57	4.35
	$28, 3, 1, 20$	18.19	16.4	4.16	3.16
	$28, 5, 1, 20$	34.45	15.8	6.61	4.22
$\textsf{MatMul}_{m,n,v}$	$1, 100, 1$	7.34	8.3	0.064	0.032
	$1, 500, 100$	52.5	21.35	16.17	8.1
	$784, 128, 10$	114.9	39.2	33.77	17.51
Maxpool	$8 \times 8 \times 50, 4 \times 4$	609.7	501.8	25.92	21.51
	$24 \times 24 \times 16, 2 \times 2$	654.7	476.9	59.71	49.85
	$24 \times 24 \times 20, 2 \times 2$	776.6	593	74.64	62.32
DMP	$8 \times 8 \times 50, 4 \times 4$	27.7	21.24	3.07	3.07
	$24 \times 24 \times 16, 2 \times 2$	53.4	48.3	8.89	8.89
	$24 \times 24 \times 20, 2 \times 2$	61.9	53.9	11.11	11.11
DReLU	64×16	111.3	89.9	7.2	6.1
	128×128	1180	848.3	115.34	97.64
	576×20	870	536.2	81.1	68.65
ReLu	64×16	4.29	4.13	0.819	0.368
	128×128	31.4	30.9	13.1	5.89
	576×20	25.07	19.11	9.21	4.14

Table 9: Microbenchmarks for various ML functionality protocols in LAN setting.

Acknowledgments

We thank Sai Lakshmi Bhavana Obbtattu 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.

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] NTL Library. `http://www.shoup.net/ntl/`. Accessed: 2017-09-26.

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

[6] 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 (General Data Protection Regulation). *Official Journal of the European Union*, L119, May 2016.

[7] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 420–432, 1991.

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

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 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 *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[12] Ran Canetti. Security and composition of multiparty cryptographic protocols. In *Journal of CRYPTOLOGY*, 2000.

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

[14] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, pages 182–199. Springer, 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] Herv 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] Melissa Chase, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, and Peter Rindal. Private collaborative neural network learning. Cryptology ePrint Archive, Report 2017/762, 2017. `https://eprint.iacr.org/2017/762`.

[18] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, 1988.

[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 *International Journal of Applied Cryptography*, 2008.

[21] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

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

[23] 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 *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016.

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[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 Symposium on Theory of Computing (STOC)*, 1987.

[26] E. Hesamifard, H. Takabi, and M. Ghasemi. CryptoDL: Deep Neural Networks over Encrypted Data. *ArXiv e-prints*, 2017.

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. *CoRR*, abs/1801.05507, 2018.

[28] 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.

[29] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Advances in Cryptology—CRYPTO*, 2000.

[30] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[31] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[32] Michael A. Nielsen. Neural networks and deep learning. 2015.

[33] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 801–812, 2013.

[34] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography*, 2007.

[35] 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. Cryptology ePrint Archive, Report 2017/1164, 2017. `https://eprint.iacr.org/2017/1164`.

[36] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. *IACR Cryptology ePrint Archive*, 2017:502, 2017.

[37] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1310–1321, 2015.

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

[39] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.

[40] Xiangxin Zhu, Carl Vondrick, Charless Fowlkes, and Deva Ramanan. Do we need more training data? 2016.

A Other Related Work

In recent years, privacy preserving machine learning has received considerable attention from the research community [29, 11, 23, 31, 30]. Research in this domain can be broadly divided into (1) privacy preserving classification (2) privacy preserving training. Since we already discussed and compared our work against [31, 30, 35, 36, 27], here we focus on other related works.

Privacy preserving classification. 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, [38] gave an improved protocol for prediction using decision trees. The work of Gilad-Barach *et al.* [23] show how to perform secure neural network prediction over encrypted data using homomorphic encryption techniques, by approximating the ReLU activation function to a quadratic function that is more "friendly" towards homomorphic encryption schemes. Since this approximation results in loss in accuracy, there have been works that approximate using higher degree polynomials [16], but incur higher cost.

Privacy preserving training. The problem of secure training on joint data is much more challenging problem. Perhaps the first work to consider this was by Lindell and Pinkas [29] that provided algorithms to execute decision tree based training over shared data. Nikolaenko et al. [33] implemented a secure matrix factorization to train a movie recommender system. Shokri and Smatikov [37] considered the problem of secure 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 effect of leakage occurring in this scheme is well-understood and hence, no formal security guarantee is obtained. More recently, as mentioned earlier, Mohassel and Zhang [31] show protocols for linear regression, logistic regression and neural networks in the 2-server model. Hesamifard *et al.* [26] design methods to approximate activation functions such as ReLU and Sigmoid with low degree polynomials to achieve efficient homomorphic encryption schemes for training CNNs, but their protocols are quite inefficient. An orthogonal line of work builds on differential privacy [22] for privacy preserving ML applications [17].

B Preliminaries Cont'd

B.1 Threat Model and Security

We provide a very high level formulation of security in this framework and refer the reader to [13] for further details. All parties $P_1, \cdots, P_n$ (our specific focus is when $n = 3$) are modelled as non-uniform interactive Turing machines (ITMs). Honest parties are restricted to run in probabilistic polynomial time (PPT). An adversary $\mathcal{A}$, who interacts with and acts as instructed by the environment $\mathcal{Z}$, "corrupts" a fraction of the parties; in our case of $n = 3$, the adversary corrupts up to *one* of them. These corrupted parties are under the control of the adversary and the adversary can view all messages sent and received by these parties, as well as their individual random tapes, inputs and outputs (these collection of messages is referred to as the *view* of the party). However, all parties follow the protocol specification honestly (i.e., we consider honest-but-curious security). The environment receives the complete view of all adversarial parties in the interaction. At the end of the interaction, the environment outputs a single bit. The environment and the adversary are not restricted to run in probabilistic polynomial time - i.e., we provide information-theoretic security guarantees. As is standard in all information-theoretic protocols, we only assume point-to-point secure channels between all pairs of parties in the protocol and do not concern ourselves with how these channels are implemented.

We now two interactions: In the *real* interaction, the parties run a protocol Π in the presence of $\mathcal{A}$ and $\mathcal{Z}$, with input z, $z \in \{0,1\}^*$. Let $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the binary distribution ensemble

describing $\mathcal{Z}$'s output in this interaction. In the *ideal* interaction, parties send their inputs to a trusted *functionality* machine $\mathcal{F}$ that carries the desired computation truthfully. Let $\mathcal{S}$ (the *simulator*) denote the adversary in this idealized execution, and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ the binary distribution ensemble describing $\mathcal{Z}$'s output after interacting with adversary $\mathcal{S}$ and ideal functionality $\mathcal{F}$.

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 the real interaction apart from the ideal interaction, except with negligible probability (in the security parameter κ). In other words, if the two binary distribution ensembles above are statistically indistinguishable.

Finally, protocols typically invoke other sub-protocols. In this framework the *hybrid model* is like a real interaction, except that some invocations of the sub-protocols are replaced by the invocation of an instance of an ideal functionality $\mathcal{F}$; this is called the "$\mathcal{F}$-hybrid model".

B.2 Neural Network Training Algorithms

We discuss the neural networks considered in this work as well as some ML background in this section. First network that we consider is from SecureML [31] that we refer to as Network A.

Network A [31]. We follow the exact same network as in [31] – for completeness, we provide details of the network here as well. In particular, we consider a 3 layer DNN trained over the MNIST data [3]. The dimensions of the DNN are: (input) $784 \times 128 \times 128 \times 10$ (output) and each layer is fully connected. We use a standard *one hot* encoding for the output classification.

We follow the notation from [32] closely. 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 $\mathsf{ReLU}(x) = \mathsf{max}(0, x)$. Usually, the *softmax* function, defined as $\mathsf{SM}(u_i) = \frac{e^{-u_i}}{\sum e^{-u_i}}$ is applied to the output of the last layer. This function, being hard to compute cryptographically in a secure manner, is approximated by the function $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ – this is similar to what is done in the work of [31]. The idea behind the SM function is to convert the output values into a probability distribution - the same effect being also achieved by the ASM function. The backward propagation 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.

Forward/Backward Prop Equations. We denote by l a generic layer of the network, where $1 \leq l \leq L$. We use w^l_{jk} to denote the weight of the connection from k^{th} neuron in the $(l-1)^{\text{th}}$ layer to neuron j^{th} in the lth layer. We use a^l_j, b^l_j for the activation and bias of the jth neuron in the lth layer. We also define $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_k$ for notational convenience. We use y_j to denote the output.

We drop the lower indices to denote the corresponding vector/matrix – for instance, w^l denotes the weight matrix between the $(l-1)^{\text{th}}$ and l^{th} layer, whereas w^l_{jk} denote individual values. The cost function used is the cross entropy function and is given by:

$$C = -\frac{1}{n} \sum_s \sum_j \left(y_j \ln a^L_j + (1 - y_j) \ln(1 - a^L_j) \right) \tag{1}$$

where n is the number of samples and s is a generic sample. The forward propagation is governed

by the following equation:

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_k^l\right) \tag{2}$$

where σ is the non-linear operation, in our case $\mathsf{ReLU}(\cdot)$. Using $\odot$ to denote Hadamard product (element wise product) we define δ_j^l as the error of neuron j in layer l and is given by $\partial C/\partial z_j^l$. The backward propagation equations are an approximation of actual gradients given that the forward pass contains $\mathsf{ASM}(\cdot)$. The backward propagation equations are faithful to sigmoid function as the last layer activation function[6] and are given by the following 4 equations:

$$\delta^L = a^L - y \tag{3a}$$

$$\delta^l = (w^{l+1})^T \delta^{l+1} \odot \mathsf{ReLU}'(z^l) \tag{3b}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3c}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{3d}$$

Eq. (3a) computes the error of the last layer, Eq. (3b) gives a way of computing the errors for layer l in terms of the errors for layer $l+1$, the weights w^{l+1} and z^l. Finally, Eq. (3c) and (3d) give compute the gradients of the biases and weights respectively.

Stochastic Gradient Descent (SGD). SGD is an iterative algorithm to minimize a function. We use SGD to train our DNN by initializing the weights to random values. In the *forward pass*, the network propagates from the inputs a^1 to compute the output y and in the *backward pass* the gradients are computed and the weights are updated. For efficiency reasons, instead of computing the forward and backward pass on each data sample, frequently a small set of samples are chosen randomly (called a *mini-batch*) and propagated together. The size of the mini-batch is denoted by B, set to 128 in this work. The complete algorithm for the 3-layer neural network is described in Algorithm 11.

Other networks. The other three networks we consider are as follows:

Network B [30] 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. Back propagation equations are computed appropriately. This network, after training for 15 epochs, provides an inference accuracy of 98.77% on the MNIST dataset.
Network C [28]. 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

[6] sigmoid function is given by $f(x) = 1/(1 + e^{-x})$

Algorithm 11 DNN Π_{ML}:

Input: Inputs are read into a^1 one mini-batch at a time.

1: **for** $l = 2 : L$ **do**
2: $\quad z^{x,l} = w^l a^{x,l-1} + b^l$
3: $\quad a^{x,l} = \sigma(z^{x,l})$
4: **end for**
5: $\mathsf{ASM}(a_i^L) = \frac{\mathsf{ReLU}(a_i^L)}{\sum \mathsf{ReLU}(a_i^L)}$
6: $\delta^{x,L} = \mathsf{ASM}(a^{x,L}) - y^x$
7: **for** $l = L-1 : 2$ **do**
8: $\quad \delta^{x,l} = w^{l+1}\delta^{x,l+1} \odot \mathsf{ReLU}'(z^{x,l})$
9: **end for**
10: **for** $l = L : 2$ **do**
11: $\quad b^l \to b^l - \frac{\alpha}{|B|}\sum_x \delta^{x,l}$
12: $\quad w^l \to w^l - \frac{\alpha}{|B|}\sum_x \delta^{x,l}(a^{x,l-1})^T$
13: **end for**

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. This network, after training for 15 epochs, provides an inference accuracy of 99% on the MNIST dataset.

Network D [35]. 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. Chameleon [35] claims that this network gives accuracy of 99%.

C 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 the methodology of [31] and describe details below.

C.1 Number Encoding

We use fixed point arithmetic to perform all the computations required by the DNN. In other words, all numbers are represented as integers in the `uint64_t` native C++ datatype. We use a precision of $l_D = 13$ bits for representing all numbers. In other words, 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 popularity in the systems community with the introduction of fixed-point data types [2].

C.2 Decimal arithmetic

To perform decimal arithmetic in an integer ring, we use the same solution as is used in [31]. 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 [31] 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 12. We refer the reader to [31] for further details.

Algorithm 12 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 $-\Pi_{\mathsf{AS}}(-\langle X\rangle_1^L, \alpha)$.

D Functionalities

All the functionalities we describe below are 3-party functionalities. We call the functionalities directly related to high-level ML functions (such as matrix multiplication, derivative of ReLU, ReLU, Maxpool, and division) as main functionalities. For these, we will maintain the following invariant: Parties P_0, P_1 have shares over $\mathbb{Z}_L$ of certain values as input and party P_2 has no input. At the end, P_0, P_1 end up with shares over $\mathbb{Z}_L$ of output of a certain function of the input values[7].

Our protocols also rely on some supporting functionalities such a private compare, share-convert, compute MSB, etc, that we describe after main functionalities. We note that for these supporting functionalities the above invariant may not hold.

D.1 Main functionalities

We first describe the main functionalities whose corresponding protocols can be put together to get a complete protocol for an ML algorithm.

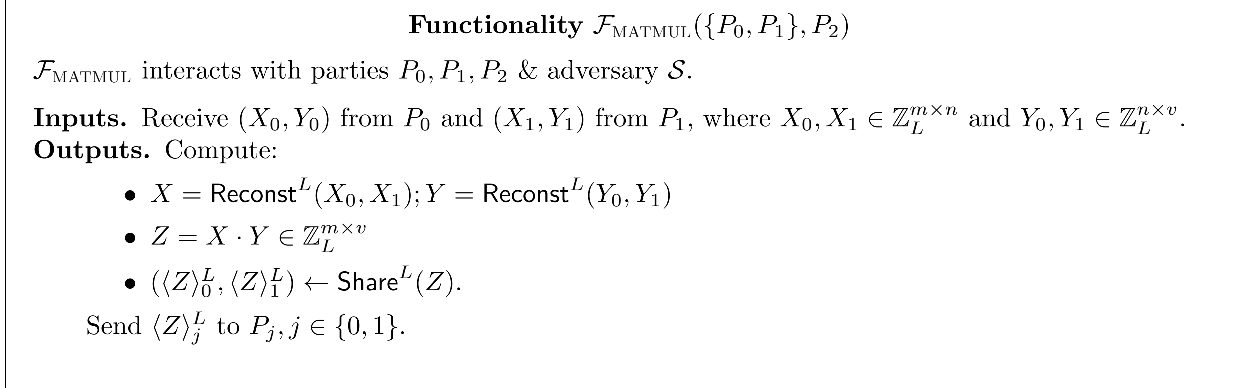
Matrix Multiplication

The first 3-party functionality that we describe computes matrix multiplication over secret shared values and secret shares the resultant product matrix amongst two parties. P_0 holds a pair (X_0, Y_0) and P_1 holds a pair (X_1, Y_1), where $X_0, X_1 \in \mathbb{Z}_L^{m\times n}$ and $Y_0, Y_1 \in \mathbb{Z}_L^{n\times v}$; P_2 has no input. The functionality computes $X = \mathsf{Reconst}^L(X_0, X_1), Y = \mathsf{Reconst}^L(Y_0, Y_1)$ and $Z := X \cdot Y \in \mathbb{Z}_L^{m\times v}$. Then, it computes $(\langle Z\rangle_0^L, \langle Z\rangle_1^L) \leftarrow \mathsf{Share}^L(Z)$. It sends $\langle Z\rangle_j^L$ to $P_j, j \in \{0, 1\}$. The functionality is described in Figure 2.

Derivative of ReLU

Parties P_0 and P_1 have values that are viewed as shares of $a \in \mathbb{Z}_L$ as input; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, computes $\mathsf{ReLU}'(a)$ which is a bit that

[7]It is trivial to extend these functionalities to the 4-party setting where P_3 also has no input and output.


Functionality $\mathcal{F}_{\text{MATMUL}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{MATMUL}}$ interacts with parties P_0, P_1, P_2 & adversary $\mathcal{S}$.

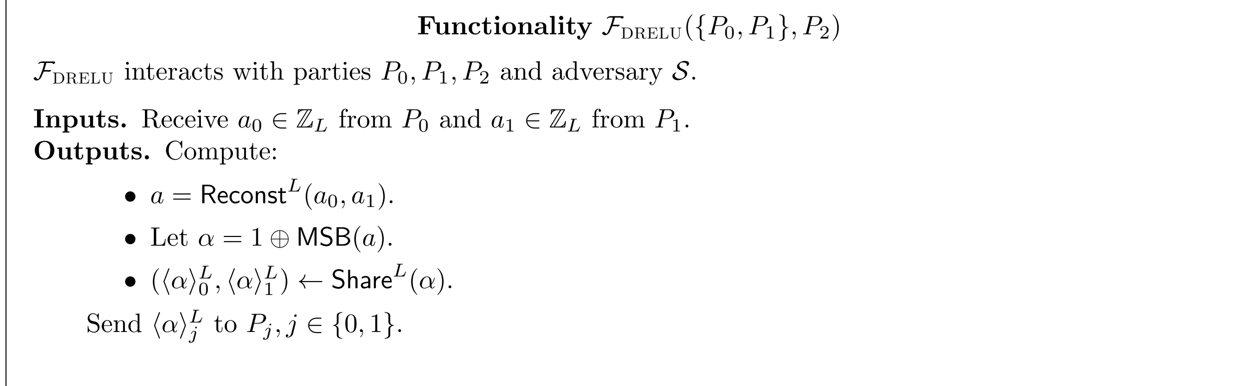
Inputs. Receive (X_0, Y_0) from P_0 and (X_1, Y_1) from P_1, where $X_0, X_1 \in \mathbb{Z}_L^{m \times n}$ and $Y_0, Y_1 \in \mathbb{Z}_L^{n \times v}$.
Outputs. Compute:

- $X = \mathsf{Reconst}^L(X_0, X_1); Y = \mathsf{Reconst}^L(Y_0, Y_1)$
- $Z = X \cdot Y \in \mathbb{Z}_L^{m \times v}$
- $(\langle Z \rangle_0^L, \langle Z \rangle_1^L) \leftarrow \mathsf{Share}^L(Z)$.

Send $\langle Z \rangle_j^L$ to $P_j, j \in \{0, 1\}$.

Figure 2: *3-party Mat. Mul. ideal functionality* $\mathcal{F}_{\text{MATMUL}}$.

is 1 if $\mathsf{MSB}(a) = 0$ and 0 otherwise. That is, $\mathsf{ReLU}'(a) = 1$ iff $a \geq 0$. It then generates shares of this bit as output to P_0 and P_1. The functionality is described in Figure 3.


Functionality $\mathcal{F}_{\text{DRELU}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{DRELU}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

Inputs. Receive $a_0 \in \mathbb{Z}_L$ from P_0 and $a_1 \in \mathbb{Z}_L$ from P_1.
Outputs. Compute:

- $a = \mathsf{Reconst}^L(a_0, a_1)$.
- Let $\alpha = 1 \oplus \mathsf{MSB}(a)$.
- $(\langle \alpha \rangle_0^L, \langle \alpha \rangle_1^L) \leftarrow \mathsf{Share}^L(\alpha)$.

Send $\langle \alpha \rangle_j^L$ to $P_j, j \in \{0, 1\}$.

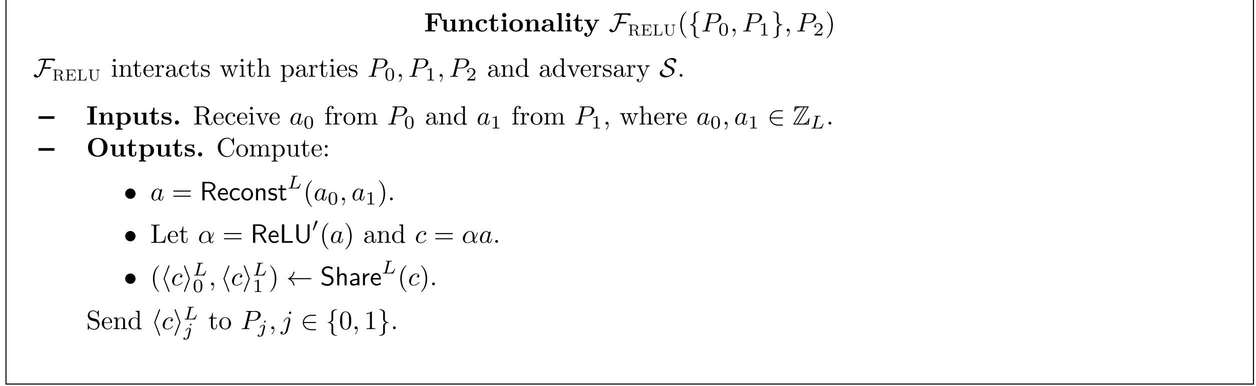
Figure 3: *3-party* ReLU' *ideal functionality* $\mathcal{F}_{\text{DRELU}}$.

ReLU

Parties P_0 and P_1 have values that are viewed as shares of $a \in \mathbb{Z}_L$ as input; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs a, computes $\mathsf{ReLU}(a)$ (which is a if $\mathsf{ReLU}'(a) = 1$ and 0 otherwise) and then generates shares of this value (over $\mathbb{Z}_L$) as output to P_0 and P_1. The functionality is described in Figure 4.

Division

Parties P_0 and P_1, as input, have values that are viewed as shares of $x, y \in \mathbb{Z}_L$; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs x and y. It computes $z = x/y$ (which is defined to be $\lfloor \frac{x}{y} \rfloor$, where x and $y, y \neq 0$ are non-negative integers) and then generates shares of z (over $\mathbb{Z}_L$) as output to P_0 and P_1. The functionality is described in Figure 5.



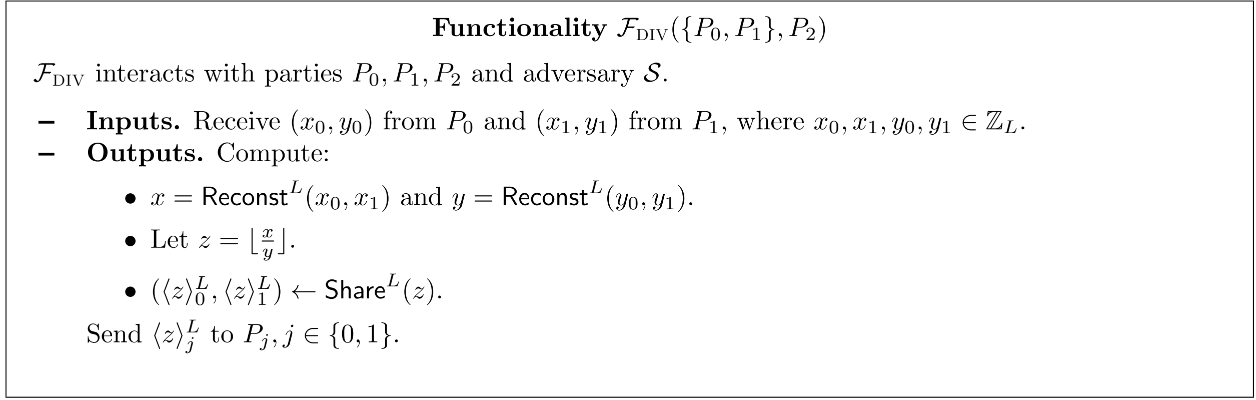
Functionality $\mathcal{F}_{\text{RELU}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{RELU}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

- **Inputs.** Receive a_0 from P_0 and a_1 from P_1, where $a_0, a_1 \in \mathbb{Z}_L$.
- **Outputs.** Compute:
  - $a = \mathsf{Reconst}^L(a_0, a_1)$.
  - Let $\alpha = \mathsf{ReLU}'(a)$ and $c = \alpha a$.
  - $(\langle c \rangle_0^L, \langle c \rangle_1^L) \leftarrow \mathsf{Share}^L(c)$.

  Send $\langle c \rangle_j^L$ to $P_j, j \in \{0, 1\}$.

Figure 4: *3-party* ReLU *ideal functionality* $\mathcal{F}_{\text{RELU}}$.



Functionality $\mathcal{F}_{\text{DIV}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{DIV}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

- **Inputs.** Receive (x_0, y_0) from P_0 and (x_1, y_1) from P_1, where $x_0, x_1, y_0, y_1 \in \mathbb{Z}_L$.
- **Outputs.** Compute:
  - $x = \mathsf{Reconst}^L(x_0, x_1)$ and $y = \mathsf{Reconst}^L(y_0, y_1)$.
  - Let $z = \lfloor \frac{x}{y} \rfloor$.
  - $(\langle z \rangle_0^L, \langle z \rangle_1^L) \leftarrow \mathsf{Share}^L(z)$.

  Send $\langle z \rangle_j^L$ to $P_j, j \in \{0, 1\}$.

Figure 5: *3-party Division ideal functionality* $\mathcal{F}_{\text{DIV}}$.

Maxpool

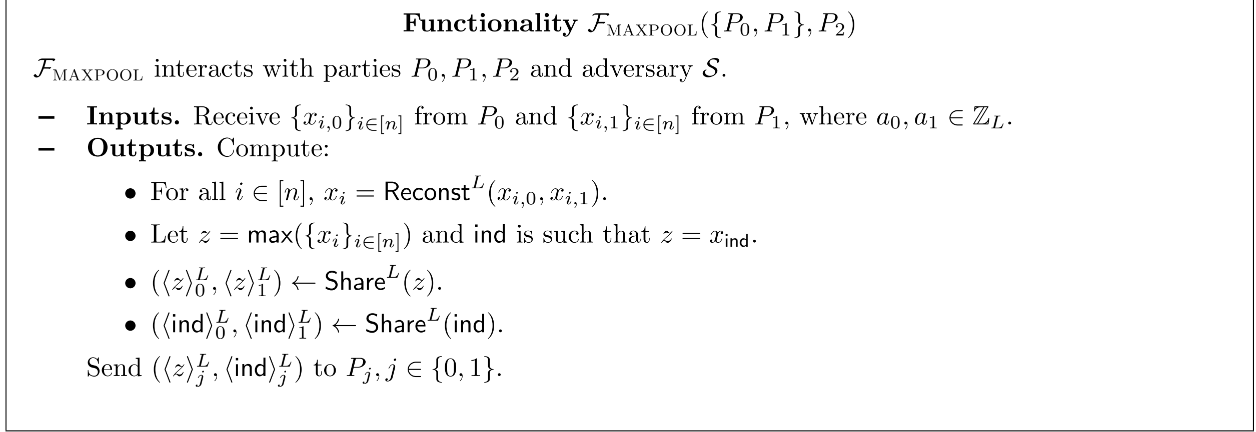
Parties P_0 and P_1, as input, have n values each that are viewed as shares of $\{x_i\}_{i \in [n]} \in \mathbb{Z}_L$; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs each x_i. It computes $z = \mathsf{max}(\{x_i\}_{i \in [n]})$ and ind is the index such that $z = x_{\mathsf{ind}}$. It then generates shares of z and ind (over $\mathbb{Z}_L$) as output to P_0 and P_1. The functionality is described in Figure 6.

Derivative Maxpool

Parties P_0 and P_1, as input, have n values each that are viewed as shares of $\{x_i\}_{i \in [n]} \in \mathbb{Z}_L$; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs each x_i. It computes $z = \mathsf{max}(\{x_i\}_{i \in [n]})$ and ind is the index such that $z = x_{\mathsf{ind}}$. Then, it creates a vector $\{y_i\}_{i \in [n]}$ such that $y_i = 1$ if $i = \mathsf{ind}$, else $y_i = 0$. It then generates shares of y_i as output to P_0 and P_1. The functionality is described in Figure 7.

D.2 Supporting functionalities

In this section, we describe some supporting functionalities



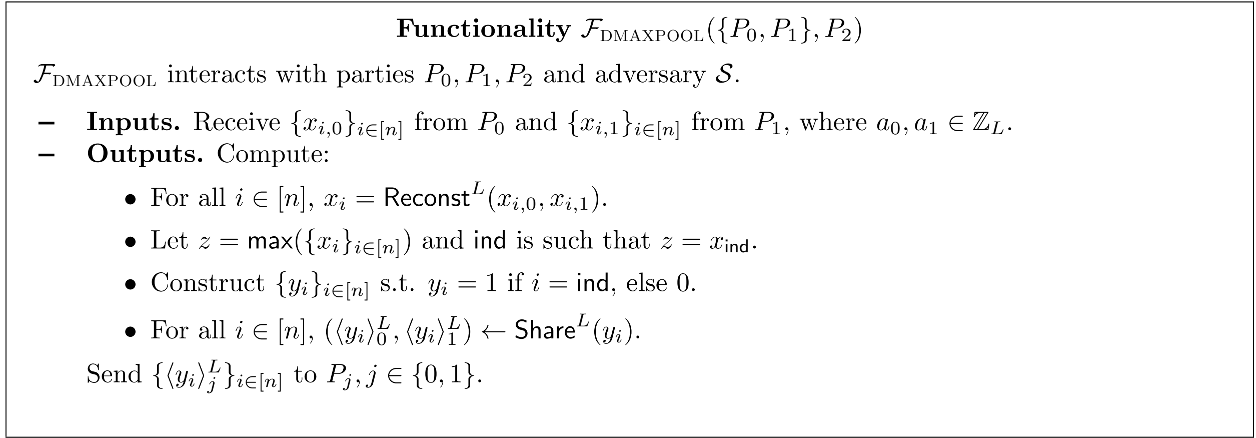
Functionality $\mathcal{F}_{\text{MAXPOOL}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{MAXPOOL}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

- **Inputs.** Receive $\{x_{i,0}\}_{i \in [n]}$ from P_0 and $\{x_{i,1}\}_{i \in [n]}$ from P_1, where $a_0, a_1 \in \mathbb{Z}_L$.
- **Outputs.** Compute:
  - For all $i \in [n]$, $x_i = \mathsf{Reconst}^L(x_{i,0}, x_{i,1})$.
  - Let $z = \mathsf{max}(\{x_i\}_{i \in [n]})$ and ind is such that $z = x_{\mathsf{ind}}$.
  - $(\langle z \rangle_0^L, \langle z \rangle_1^L) \leftarrow \mathsf{Share}^L(z)$.
  - $(\langle \mathsf{ind} \rangle_0^L, \langle \mathsf{ind} \rangle_1^L) \leftarrow \mathsf{Share}^L(\mathsf{ind})$.

  Send $(\langle z \rangle_j^L, \langle \mathsf{ind} \rangle_j^L)$ to $P_j, j \in \{0, 1\}$.

Figure 6: *3-party Maxpool ideal functionality* $\mathcal{F}_{\text{MAXPOOL}}$.



Functionality $\mathcal{F}_{\text{DMAXPOOL}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{DMAXPOOL}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

- **Inputs.** Receive $\{x_{i,0}\}_{i \in [n]}$ from P_0 and $\{x_{i,1}\}_{i \in [n]}$ from P_1, where $a_0, a_1 \in \mathbb{Z}_L$.
- **Outputs.** Compute:
  - For all $i \in [n]$, $x_i = \mathsf{Reconst}^L(x_{i,0}, x_{i,1})$.
  - Let $z = \mathsf{max}(\{x_i\}_{i \in [n]})$ and ind is such that $z = x_{\mathsf{ind}}$.
  - Construct $\{y_i\}_{i \in [n]}$ s.t. $y_i = 1$ if $i = \mathsf{ind}$, else 0.
  - For all $i \in [n]$, $(\langle y_i \rangle_0^L, \langle y_i \rangle_1^L) \leftarrow \mathsf{Share}^L(y_i)$.

  Send $\{\langle y_i \rangle_j^L\}_{i \in [n]}$ to $P_j, j \in \{0, 1\}$.

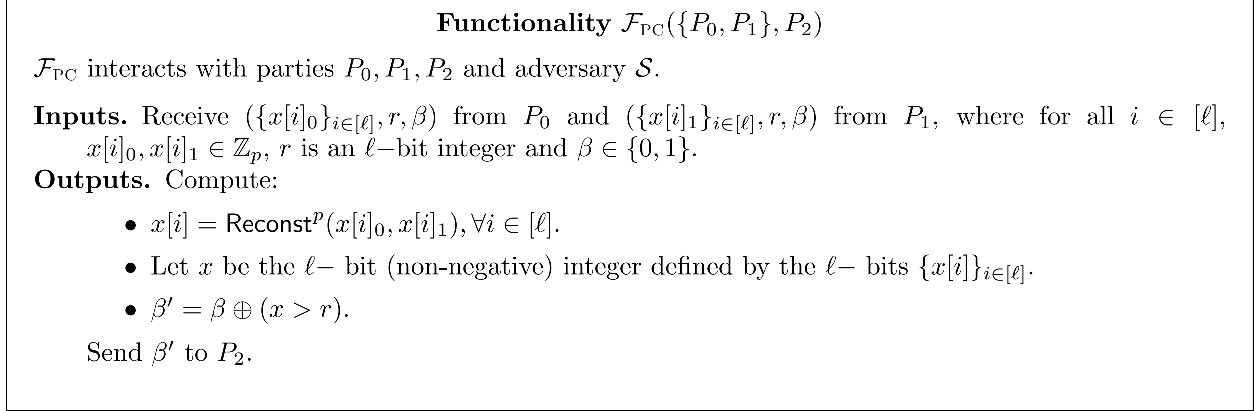
Figure 7: *3-party Maxpool ideal functionality* $\mathcal{F}_{\text{DMAXPOOL}}$.

Private Compare

It is a 3-party functionality such that parties P_0 and P_1 have values in $\mathbb{Z}_p$ that are viewed as shares of bits of an $\ell-$bit value x, a common $\ell-$bit value r and a bit β as input; P_2 has no input. Define $(x > r)$ to be the bit that is 1 if $x > r$ and 0 otherwise. The functionality takes all the above values as input from P_0 and P_1 and gives $\beta' = \beta \oplus (x > r)$ to P_2 as output. The functionality is described in Figure 8.

Share Convert

Parties P_0 and P_1 have values in $\mathbb{Z}_L$ as input, that are viewed as shares of a value $a \in \mathbb{Z}_L$ $(a \neq L - 1)$; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs a, generates fresh shares of a over $\mathbb{Z}_{L-1}$ and gives $\langle a \rangle_0^{L-1}$ to P_0 and $\langle a \rangle_1^{L-1}$ to P_1 as output. The functionality is described in Figure 9.



Functionality $\mathcal{F}_{\text{PC}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{PC}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

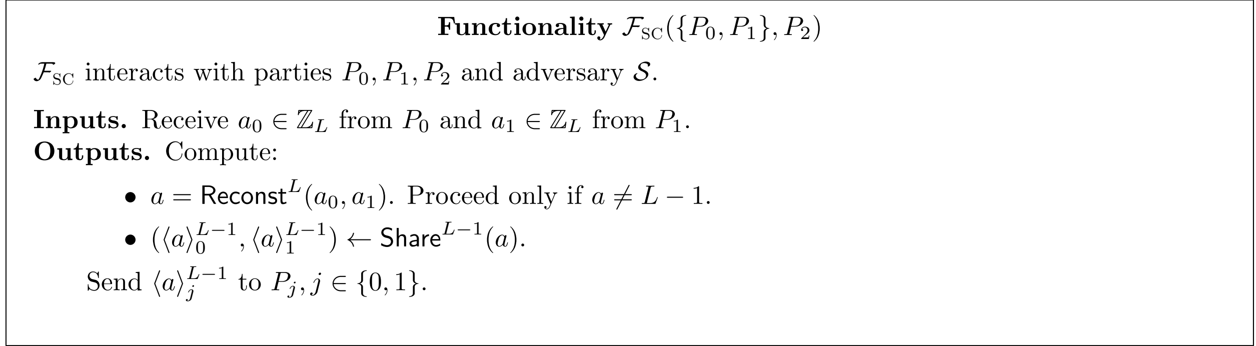
Inputs. Receive $(\{x[i]_0\}_{i \in [\ell]}, r, \beta)$ from P_0 and $(\{x[i]_1\}_{i \in [\ell]}, r, \beta)$ from P_1, where for all $i \in [\ell]$, $x[i]_0, x[i]_1 \in \mathbb{Z}_p$, r is an $\ell-$bit integer and $\beta \in \{0, 1\}$.

Outputs. Compute:

- $x[i] = \mathsf{Reconst}^p(x[i]_0, x[i]_1), \forall i \in [\ell]$.
- Let x be the $\ell-$ bit (non-negative) integer defined by the $\ell-$ bits $\{x[i]\}_{i \in [\ell]}$.
- $\beta' = \beta \oplus (x > r)$.

Send β' to P_2.

Figure 8: *3-party Private Compare ideal functionality* $\mathcal{F}_{\text{PC}}$.



Functionality $\mathcal{F}_{\text{SC}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{SC}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

Inputs. Receive $a_0 \in \mathbb{Z}_L$ from P_0 and $a_1 \in \mathbb{Z}_L$ from P_1.

Outputs. Compute:

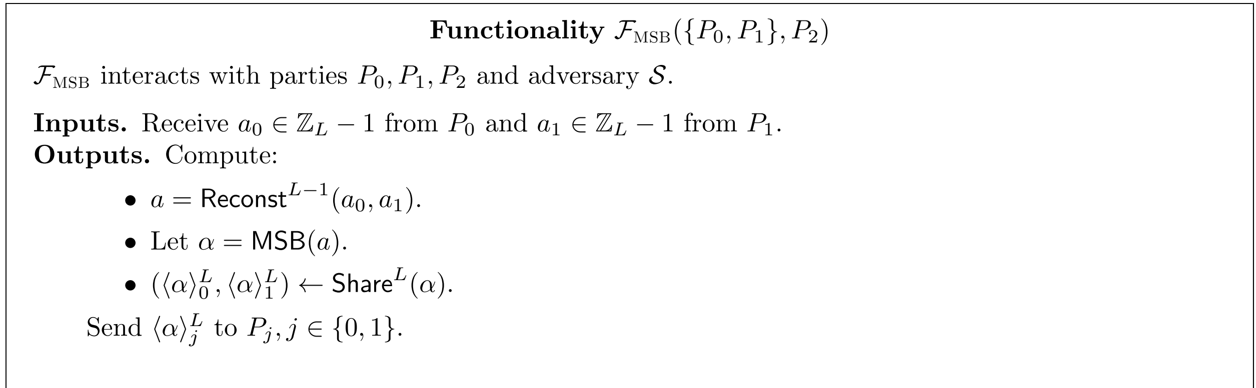
- $a = \mathsf{Reconst}^L(a_0, a_1)$. Proceed only if $a \neq L - 1$.
- $(\langle a \rangle_0^{L-1}, \langle a \rangle_1^{L-1}) \leftarrow \mathsf{Share}^{L-1}(a)$.

Send $\langle a \rangle_j^{L-1}$ to $P_j, j \in \{0, 1\}$.

Figure 9: *3-party Share Convert ideal functionality* $\mathcal{F}_{\text{SC}}$.

Compute MSB

Parties P_0 and P_1, as input, have values that are viewed as shares over $\mathbb{Z}_{L-1}$ of $a \in \mathbb{Z}_{L-1}$; P_2 has no input. The functionality reconstructs a, computes the MSB of a and then generates shares of this bit as output to P_0 and P_1. The functionality is described in Figure 10.



Functionality $\mathcal{F}_{\text{MSB}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{MSB}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

Inputs. Receive $a_0 \in \mathbb{Z}_L - 1$ from P_0 and $a_1 \in \mathbb{Z}_L - 1$ from P_1.

Outputs. Compute:

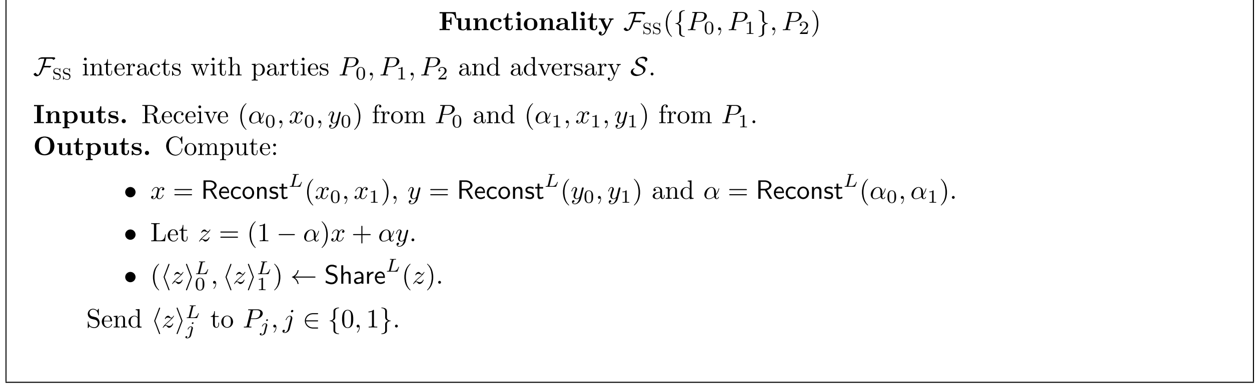
- $a = \mathsf{Reconst}^{L-1}(a_0, a_1)$.
- Let $\alpha = \mathsf{MSB}(a)$.
- $(\langle \alpha \rangle_0^L, \langle \alpha \rangle_1^L) \leftarrow \mathsf{Share}^L(\alpha)$.

Send $\langle \alpha \rangle_j^L$ to $P_j, j \in \{0, 1\}$.

Figure 10: *3-party MSB ideal functionality* $\mathcal{F}_{\text{MSB}}$.

Select Share

Parties P_0 and P_1 hold a pair of values that are viewed as shares of $x, y \in \mathbb{Z}_L$ as well as values that are viewed as shares of $\alpha \in \{0,1\}$ as input; P_2 has no input. The functionality takes the shares as input from P_0 and P_1, reconstructs x, y, α. It generates shares of x if α is 0, else shares of y and outputs to P_0 and P_1. The functionality is described in Figure 11.



Functionality $\mathcal{F}_{\text{SS}}(\{P_0, P_1\}, P_2)$

$\mathcal{F}_{\text{SS}}$ interacts with parties P_0, P_1, P_2 and adversary $\mathcal{S}$.

Inputs. Receive (α_0, x_0, y_0) from P_0 and (α_1, x_1, y_1) from P_1.
Outputs. Compute:

- $x = \mathsf{Reconst}^L(x_0, x_1)$, $y = \mathsf{Reconst}^L(y_0, y_1)$ and $\alpha = \mathsf{Reconst}^L(\alpha_0, \alpha_1)$.
- Let $z = (1-\alpha)x + \alpha y$.
- $(\langle z\rangle_0^L, \langle z\rangle_1^L) \leftarrow \mathsf{Share}^L(z)$.

Send $\langle z\rangle_j^L$ to $P_j, j \in \{0,1\}$.

Figure 11: *3-party Select Share ideal functionality* $\mathcal{F}_{\text{SS}}$.

E Remaining protocols and security proofs for 3-party

Here, we provide the protocols skipped in Section 3 and Section 4 for the 3-party case and corresponding security proofs. We also provide the proof of optimized 4-party matrix multiplication.

3-party Matrix Multiplication

Proof of Lemma 1: 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 $Z_0 + Z_1 = \big(\langle X\rangle_0^L \cdot F + E \cdot \langle Y\rangle_0^L + \langle C\rangle_0^L + U_0\big) + \big(-E \cdot F + \langle X\rangle_1^L \cdot F + E \cdot \langle Y\rangle_1^L + \langle C\rangle_1^L + U_1\big) = -E \cdot F + X \cdot F + E \cdot Y + C = -(X - A)\cdot(Y - B) + X \cdot (Y - B) + (X - A) \cdot Y + A \cdot B = X \cdot Y$.

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

Proof of Lemma 2: 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). Now, $P_j, j \in \{0,1\}$ 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 final output in Step 3 as the respective shares are randomized by u_j.

Private Compare

Proof of Lemma 3: 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[8] 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 zero.

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 randomly distributed over $\mathbb{Z}_p^*$ since the s_i values are random in $\mathbb{Z}_p^*$.

Share Convert

Proof of Lemma 4: 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

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

$x \xleftarrow{\$} \mathbb{Z}_L$ 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.

Compute MSB

Proof of Lemma 5: 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 $\mathcal{F}_{\text{PC}}$ such 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.

4-Party Matrix Multiplication

Proof of Lemma 6: We first prove the correctness of our protocol, i.e. $\mathsf{Reconst}^L(Z_0, Z_1) = XY$. To see this, observe that $\langle Z\rangle_0^L + \langle Z\rangle_1^L = \sum_{j=0,1} \langle X\rangle_j^L \cdot \langle Y\rangle_j^L + \langle W\rangle_j^L + U_j = \sum_{j=0,1} \langle X\rangle_j^L \cdot \langle Y\rangle_j^L + \langle W\rangle_j^L$ (as $U_0 + U_1 = 0^{m\times v}$). Now, $\sum_{j=0,1} \langle W\rangle_j^L = \sum_{j=0,1} \langle X\rangle_j^L \cdot \langle Y\rangle_{1-j}^L$ (since $V_0 + V_1 = 0^{m\times v}$). Therefore, $\langle Z\rangle_0^L + \langle Z\rangle_1^L = \sum_{j=0,1} \langle X\rangle_j^L \cdot \langle Y\rangle_j^L + \langle X\rangle_j^L \cdot \langle Y\rangle_{1-j}^L = (\langle X\rangle_0^L + \langle X\rangle_1^L) \cdot (\langle Y\rangle_0^L + \langle Y\rangle_1^L) = X \cdot Y$.

We first prove security of our protocol against corruption of either P_2 or P_3. Observe that P_2 and P_3 only observe $(\langle X\rangle_0^L, \langle Y\rangle_1^L)$ and $(\langle X\rangle_1^L, \langle Y\rangle_0^L)$, resp., which are fresh shares of X and Y and therefore, reveal no information about X or Y. Hence, these messages can be simulated by simply sending a pair of random matrices in $(\mathbb{Z}_L^{m\times n}, \mathbb{Z}_L^{n\times v})$. These are the only messages that P_2 and P_3 observe.

Now, we prove security against corruption of either P_0 or P_1. Party $P_j, j \in \{0,1\}$ receive $\langle W\rangle_j^L$ respectively. However, these are fresh shares of $\langle X\rangle_0^L \cdot \langle Y\rangle_1^L + \langle X\rangle_1^L \cdot \langle Y\rangle_0^L$ as they have been randomized by the random V_j matrix, respectively. Hence, they contain no information about X and Y and can be simulated by sending a random matrix in $\mathbb{Z}_L^{m\times v}$.

Finally, P_j outputs $\langle Z\rangle_j^L = \langle X\rangle_j^L \cdot \langle Y\rangle_j^L + \langle W\rangle_j^L + U_j$, which 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.

Derivative of ReLU

Proof of Lemma 7: First, we prove the correctness of our protocol, 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.

To see the security, first observe 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

Proof of Lemma 8: First, we prove the correctness of our protocol, 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[9]— 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 $c = \alpha\cdot a$.

To argue security, 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 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.

Division

The formal protocol description is provided in Algorithm 13.

Lemma 11. *Protocol $\Pi_{\mathsf{DIV}}(\{P_0,P_1\},P_2)$ in Algorithm 13 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 ith iteration we compute the $q[i]$, the ith 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, $\mathsf{ReLU}'(z_m)=1$. By correctness[10] 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 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the final output in Step 9 as the respective shares are randomized by s_j. □

[9]When we instantiate the functionality $\mathcal{F}_{\text{DRELU}}$ using protocol Π_{DRELU}, we would ensure that the conditions of Lemma 7 are met.

[10]When we instantiate the functionality $\mathcal{F}_{\text{DRELU}}$ using protocol Π_{DRELU}, we would ensure that the conditions of Lemma 7 are met.

Algorithm 13 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 call $\mathcal{F}_{\text{DRELU}}(\{P_0, P_1\}, P_2)$ 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 $(\langle \beta_i\rangle_j^L, \langle 2^i\rangle_j^L)$ and P_0, P_1 learn $\langle k_i\rangle_0^L$ and $\langle k_i\rangle_1^L$, resp.
6: 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.
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 $\langle q\rangle_j^L = \sum_{i=0}^{\ell-1}\langle k_i\rangle_j^L + s_j$.

Maxpool

Proof of Lemma 9: 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 the value 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 [11] 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 algorithm in the general case.

Proof of Lemma 10: Let k^* be the index of the maximum value and E_r denote the unit vector with 1 in the rth 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.

[11] When we instantiate the functionality $\mathcal{F}_{\text{DRELU}}$ using protocol Π_{DRELU}, we would ensure that the conditions of Lemma 7 are met.

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 $n|L$, 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 $\langle D \rangle_j^L = (\langle E^{(-g \bmod n)} \rangle_j^L, \langle E^{(1-g \bmod n)} \rangle_j^L, \cdots, \langle E^{(n-1-g \bmod n)} \rangle_j^L)$, $\langle D^{(k-g) \bmod n} \rangle_0^L$ 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 kth position and 0 everywhere else. E_0 denotes the all zeroes vector. Details are presented in Algorithm 14.

Algorithm 14 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 $\langle \mathsf{DMP}_n \rangle_j^L + U_j$.

Lemma 12. *Protocol $\Pi_{\mathsf{DMP}}(\{P_0, P_1\}, P_2)$ in Algorithm 14 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 14. This proof is nearly identical to the proof of correctness of Algorithm 9. As before, we

prove this by induction and 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.

$\square$