Proceedings on Privacy Enhancing Technologies ; 2021 (1):188–208

Sameer Wagh*, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin

FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning

Abstract: We propose FALCON, an end-to-end 3-party protocol for efficient private training and inference of large machine learning models. FALCON presents four main advantages – (i) It is highly *expressive* with support for high capacity networks such as VGG16 (ii) it supports batch normalization which is important for training complex networks such as AlexNet (iii) FALCON guarantees *security with abort* against malicious adversaries, assuming an honest majority (iv) Lastly, FALCON presents new theoretical insights for protocol design that make it *highly efficient* and allow it to outperform existing secure deep learning solutions. Compared to prior art for private inference, we are about $8\times$ faster than SecureNN (PETS'19) on average and comparable to ABY^3 (CCS'18). We are about $16-200\times$ more communication efficient than either of these. For private training, we are about $6\times$ faster than SecureNN, $4.4\times$ faster than ABY^3 and about $2-60\times$ more communication efficient. Our experiments in the WAN setting show that over large networks and datasets, *compute operations* dominate the overall latency of MPC, as opposed to the communication.

Keywords: Multi Party Computation, Secure Comparison, Deep Learning, Neural Networks

DOI 10.2478/popets-2021-0011
Received 2020-05-31; revised 2020-09-15; accepted 2020-09-16.

1 Introduction

With today's digital infrastructure, tremendous amounts of private data is continuously being generated – data which combined with deep learning algorithms can transform the current social and technological landscape. For example, distribution of child exploitative imagery has plagued social media platforms [1, 2]. However, stringent government regulations hamper automated detection of such harmful content. Support for secure computation of state-of-the-art image classification networks would aid in detecting child exploitative imagery on social media. Similarly, there is promise in analyzing medical data across different hospitals especially for the treatment of rare diseases [3]. In both these scenarios, multiple parties (i.e., social media platforms or hospitals) could co-operate to train efficient models that have high prediction accuracy. However, the sensitive nature of such data demands deep learning frameworks that allow training on data aggregated from multiple entities while ensuring strong privacy and confidentiality guarantees. A synergistic combination of secure computing primitives with deep learning algorithms would enable sensitive applications to benefit from the high prediction accuracies of neural networks.

Secure multi-party computation (MPC) techniques provide a transformative capability to perform secure analytics over such data [4, 19, 20]. MPC provides a cryptographically secure framework for computations where the collaborating parties do not reveal their secret data to each other. The parties only learn the output of the computation on the combined data while revealing nothing about the individual secret inputs. Recently, there has been research in reducing the performance overhead of MPC protocols, specifically tailored for machine learning [5, 8, 9, 11, 12]. In this work, we focus on advancing the research in other dimensions such as expressiveness, scalability to millions of parameters, and stronger security guarantees (where parties can arbitrarily deviate from the protocol) that are necessary for practical deployment of secure deep learning frameworks. We present FALCON — an efficient and expressive 3-party deep learning framework that provides support for both training and inference with malicious security guarantees. Table 1 provides a detailed comparison of FALCON with prior work.

Contributions. Our contributions are as follows:

***Corresponding Author: Sameer Wagh:** Princeton University & UC Berkeley, E-mail: swagh@princeton.edu
Shruti Tople: Microsoft Research, E-mail: shruti.tople@microsoft.com
Fabrice Benhamouda: Algorand Foundation, E-mail: fabrice.benhamouda@normalesup.org
Eyal Kushilevitz: Technion, E-mail: eyalk@cs.technion.ac.il
Prateek Mittal: Princeton University, E-mail: pmittal@princeton.edu
Tal Rabin: Algorand Foundation, E-mail: talrny@yahoo.com

		Theoretical Metrics												Evaluation Metrics										
		Private Capability		Threat Model		Supported Layers					Techniques Used			LAN/ WAN		Evaluation Dataset			Network Architectures					
	Framework	*Inference*	*Training*	*Semi-honest*	*Malicious*	*Linear*	*Convolution*	*Exact ReLU*	*Maxpool*	*Batch-Norm*	*HE*	*GC*	*SS*	*LAN*	*WAN*	*MNIST*	*CIFAR-10*	*Tiny ImageNet*	*From [4]*	*From [5]*	*From [6]*	*LeNet*	*AlexNet*	*VGG-16*
2PC	MiniONN [6]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	●	●	○	●	○	●	○	○	○
	Chameleon [5]	●	○	●	○	●	●	●	●	○	○	●	●	●	●	●	●	○	●	●	●	○	◐	○
	EzPC [7]	●	○	●	○	●	●	●	●	○	○	●	●	●	●	●	●	○	●	○	●	○	○	○
	Gazelle [8]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	●	●	○	●	●	●	○	○	○
	SecureML [4]	●	●	●	○	●	●	●	●	○	●	●	●	●	◐	●	○	○	●	○	○	○	○	○
	XONN [9]	●	○	●	●	●	●	●	●	◐	○	●	●	●	○	●	●	○	●	●	●	○	○	◐
	Delphi [10]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	○	●	○	○	○	●	○	○	◐
3PC	ABY^3 [11]	●	●	●	●	●	●	●	●	○	○	●	●	●	◐	●	○	○	●	●	●	○	○	○
	SecureNN [12]	●	●	●	◐	●	●	●	●	◐	○	○	●	●	●	●	○	○	●	●	●	○	○	○
	CryptFlow [13]	●	○	●	●	●	●	●	●	○	○	○	●	◐	○	●	●	●	●	●	●	●	○	◐
	QuantizedNN [14]*	●	○	●	●	●	●	●	●	◐	◐	○	●	●	●	○	○	●	○	○	○	○	○	◐
	ASTRA [15]	●	●	●	◐	●	●	●	●	○	○	●	●	●	●	●	○	○	●	○	○	○	○	○
	BLAZE [16]	●	●	●	●	●	●	●	●	○	○	●	●	●	●	◐	○	○	●	○	○	○	○	○
	Falcon (This Work)	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	◐	●	●	●	●	●	●
4PC	FLASH [17]	●	●	●	●	●	●	●	●	○	○	○	●	●	●	●	◐	○	●	◐	○	○	○	○
	Trident [18]	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	○	○	●	○	○	○	○	○

Table 1. Comparison of various private deep learning frameworks. FALCON proposes efficient protocols for non-linear functionalities such as ReLU and batch normalization (1) purely using modular arithmetic (2) under malicious corruptions (3) supporting both private training and inference. FALCON also provides a comprehensive evaluation (1) over larger networks and datasets (2) extensively compares with related work (3) provides newer insights for future directions of PPML. ● indicates the framework supports a feature, ○ indicates not supported feature, and ◐ refers to fair comparison difficult due to the following reasons: SecureNN provides malicious privacy but not correctness and supports division but not batch norm, XONN supports a simplified batch norm specific to a binary activation layer, ABY^3 does not present WAN results for neural networks, Chameleon evaluates over a network similar to AlexNet but using the simpler mean-pooling operations, and due to the high round complexity and communication, SecureML provides an estimate of their WAN evaluation, Delphi evaluates over network such as ResNet-32, CryptFlow evaluates networks such as DenseNet-121, ResNet-50, uses weaker network parameters in LAN and uses ImageNet dataset. QuantizedNN uses inherent quantization of the underlying NN and performs extensive evaluation over MobileNet architectures and * refers to 3PC version among the 8 protocols. BLAZE uses a Parkinson disease dataset, similar in dimension to MNIST. FLASH and Trident use few other smaller datasets in their evaluation as well as evaluate over increasing number of layers over the network architecture from [4].

(1) Malicious Security: FALCON provides strong security guarantees in an honest-majority adversarial setting. This assumption is similar to prior work where majority of the parties (e.g., 2 out of three) behave honestly [11, 21]. FALCON proposes new protocols that are secure against such corruptions and ensure that either the computation always correctly completes or aborts detecting malicious activity. We achieve this by designing new protocols for the computation of non-linear functions (like ReLU). While MPC protocols are very efficient at computing linear functions, computing non-linear functions like ReLU is much more challenging. We propose solutions both for the malicious security model and provide even more efficient protocols where semi-honest security is sufficient. We formally prove the security of FALCON using the standard simulation paradigm (see Section 4.1). We implement both the semi-honest and malicious protocols in our end-to-end framework. In this manner, FALCON provides a choice to the developers to select between either of the security guarantees depending on the trust assumption among the parties and performance requirements (improved performance for semi-honest protocols).

(2) Improved Protocols: FALCON combines techniques from SecureNN [12] and ABY^3 [11] that result in improved protocol efficiency. We improve the theoretical complexity of the central building block – derivative of ReLU – by a factor of $2\times$ through simplified algebra for fixed point arithmetic. We demonstrate our protocols in a smaller ring size, which is possible using an exact yet expensive truncation algorithm. However, this enables the entire framework to use a smaller datatype, thus reducing their communication complexity at least $2\times$. This reduced communication is critical to the communication improvements of FALCON over prior work. Furthermore, as can be seen in Section 5, these theoretical improvements lead to even larger practical improvements due to the recursive dependence of the complex functionalities on the improved building blocks. Overall, we demonstrate how to achieve maliciously secure pro-

tocols for non-linear operations using arithmetic secret sharing and avoiding the use of interconversion protocols (between arithmetic, Boolean and garbled circuits).

(3) Expressiveness: Our focus is to provide simple yet efficient protocols for the fundamental functionalities commonly used in state-of-the-art neural networks. Batch normalization, has been previously considered in privacy-preserving inference as linear transformation using Homomorphic Encryption [22–24]. However, batch normalization is critical for stable convergence of networks as well as to reduce the parameter tuning required during training of neural networks. FALCON demonstrates full support for `Batch-Normalization` layers, both forward and backward pass, in private machine learning. In other words, FALCON supports both private training and private inference. This extensive support makes FALCON expressive, thereby supporting evaluation of large networks with hundreds of millions parameters such as VGG16 [25] and AlexNet [26] over datasets such as MNIST [27], CIFAR-10 [28] as well as Tiny ImageNet [29] including in both the LAN and WAN network settings. Designing secure protocols for training is more difficult due to the operations involved in back-propagation which are not required for inference. A number of prior works assume training in a trusted environment and hence provide support for only inference service [5–9]. However, sensitive data is often inaccessible even during training as described in our motivating application in Section 2.

End-to-end Implementation and Results. We implement both the semi-honest and malicious variants of FALCON in our end-to-end framework. The codebase is written in `C++` in about 14.6k LOC and will be open sourced. We experimentally evaluate the performance overhead of FALCON for both private training and inference on multiple networks and datasets. We use 6 diverse networks ranging from simple 3-layer multi-layer perceptrons (MLP) with about $118,000$ parameters to large networks with about 16-layers having 138 million parameters. We trained these networks on MNIST [27], CIFAR-10 [28] and Tiny ImageNet [29] datasets as appropriate based on the network size. We note that FALCON is one of the few private ML frameworks to support training of high capacity networks such as AlexNet and VGG16 on the Tiny ImageNet dataset. We perform extensive evaluation of our framework in both the LAN and WAN setting as well as semi-honest and malicious adversarial setting. For private inference, we are $16\times$ faster than XONN [9], $32\times$ faster than Gazelle [8], $8\times$ faster than SecureNN, and comparable to ABY^3 on average. For private training, we are $4.4\times$ faster than ABY^3 and $6\times$ faster than SecureNN [12]. Depending on the network, our protocols can provide *up to an order of magnitude* performance improvement. FALCON is up to two orders of magnitude more communication efficient than prior work for both private training and inference. Our results in the WAN setting show that compute operations dominate the overall latency for large networks in FALCON and not the communication rounds. Hence, FALCON is an optimized 3-PC framework w.r.t. the communication which is often the bottleneck in MPC.

2 FALCON Overview

Next, we describe the application setting for FALCON, provide a motivating application, state the threat model, and an overview of our technical contributions.

2.1 A 3-Party Machine Learning Service

We consider the following scenario. There are two types of users, the first own data on which the learning algorithm will be applied, we call them data holders. The second are users who query the system after the learning period, we call these query users. These two sets of users need not be disjoint. We design a machine learning service. This service is provided by 3 parties which we call computing servers. We assume that government regulations or other social deterrents are sufficient enforcers for non-collusion between these computing servers. The service works in two phases: the training phase where the machine learning model of interest is trained on the data of the data holders and the inference phase where the trained model can be queried by the query users. The data holders share their data in a replicated secret sharing form [30] between the 3 computing servers. These 3 servers utilize the shared data and privately train the network. After this stage, query users can submit queries to the system and receive answers based on the newly constructed model held in shared form by the three servers. This way, the data holders' input has complete privacy from each of the 3 servers. Moreover, the query is also submitted in shared form and thus is kept secret from the 3 servers.

Recent advances in MPC have rendered 3PC protocols some of the most efficient protocols in the space of privacy-preserving machine learning. Though MPC is not a broadly deployed technology yet, the 3PC adversarial model has enjoyed adoption [31–33] due to their efficiency and simplicity of protocols. Below, we describe a concrete motivating application that would benefit from such a 3-party secure machine learning service.

2.1.1 Motivating Application: Detection of Child Exploitative Images Online

In recent years, the distribution of child exploitative imagery (CEI) has proliferated with the rise of social media platforms – from half a million reported in between 1998-2008 to around 12 million reports in 2017 and 45 million in 2018 [1, 2]. Given the severity of the problem and stringent laws around the handling of such incidents (18 U.S. Code §2251, 2252), it is important to develop solutions that enable efficient detection and handling of such data while complying with stringent privacy regulations. Given the success of ML, especially for image classification, it is important to leverage ML in detecting CEIs (a computer vision application). FALCON'S approach, in contrast to deployed systems such as PhotoDNA [34], enables the use of ML for this usecase. However, the inability to generate a database of the original images (due to legal regulations) leads to a problem of lack of training data. FALCON provides a cryptographically secure framework for this conundrum, where the client data is split into unrecognizable parts among a number of non-colluding entities. In this way, the solution is two-fold, MPC enables the ability to accumulate good quality training data and at the same time can enable machine-learning-as-a-service (MLaaS) for the problem of CEIs. The 3 computing parties can be Facebook, Google, and Microsoft, and will in turn be the providers of such a service. A public API can be exposed to entities willing to make use of this service, very similar to the PhotoDNA portal [34]. Organizations (clients of this service) that deal with significant number of CEI's can send automated requests to these 3 servers and locally reconstruct the classification result using the received responses. In terms of the adversarial model, we believe that the stringent legal framework around this application is a sufficient deterrent for these large organizations to prevent collusion among the parties. Similarly, a maliciously secure adversarial model further safeguards against individual servers being compromised. In this manner, MPC can enable an end-to-end solution to automated detection of CEIs in social media with strong privacy to the underlying data.

2.2 Threat Model

Our threat model assumes an honest majority among the three parties in the setting described above. This is a common adversarial setting considered in previous secure multi-party computation approaches [4, 11, 21, 30]. We consider that one of the three parties can be either semi-honest or malicious. A semi-honest adversary passively tries to learn the secret data of the other parties while a malicious adversary can arbitrarily deviate from the protocol. We assume the private keys of each of the parties are stored securely and not susceptible to leakage. We do not protect against denial of service attacks where parties refuse to cooperate. Here, FALCON simply resorts to aborting the computation.

Assumptions & Scope. The 3 parties each have shared point-to-point communication channels and pairwise shared seeds to use AES as a PRNG to generate cryptographically secure common randomness. We note that as the query users receive the answers to the queries in the clear FALCON does not guarantee protecting the privacy of the training data from attacks such as model inversion, membership inference, and attribute inference [35–37]. Defending against these attacks is an orthogonal problem and out of scope for this work. We assume that users provide consistent shares and that model poisoning attacks are out of scope.

2.3 Technical Contributions

In this section, we summarize some of the main contributions of this work with a focus on techniques used to achieve our results and improvements.

Hybrid Integration for Malicious Security. FALCON consists of a hybrid integration of ideas from SecureNN and ABY^3 along with newer protocol constructions for privacy-preserving deep learning. SecureNN does not provide correctness in the presence of malicious adversaries. Furthermore, the use of semi-honest parties in SecureNN makes it a significant challenge to convert those protocols to provide security against malicious corruptions. We use replicated secret sharing (such as in [11, 21, 30]) as our building block and use the redundancy to enforce correct behaviour in our protocols. Note that changing from the 2-out-of-2 secret sharing scheme in SecureNN to a 2-out-of-3 replicated secret sharing crucially alters some of the building blocks – these protocols are a new contribution of this work. We work in the 3 party setting where at most one party can be corrupt. We prove each building block secure in the Universal Composability (UC) framework. We show that our protocols are (1) perfectly secure in the standalone model, i.e., the distributions are identical and not just statistically close in a model where the protocol is executed only once; and (2) have straight-line black-box simulators, i.e., only assume oracle access and do no rewind. Theorem 1.2 from Kushilevitz *et al.* [38] then implies security under general concurrent composition.

Theoretical Improvements to Protocols. Falcon proposes more efficient protocols for common machine learning functionalities while providing stronger security guarantees. We achieve this through a number of theoretical improvements for reducing both the computation as well as the communication. First, in Falcon all parties execute the same protocol in contrast to SecureNN where the protocol is asymmetric. The uniformity of the parties leads to more optimal resource utilization. Second, the protocol for derivative of ReLU, in SecureNN [12] first transforms the inputs using a `Share Convert` subroutine (into secret shares modulo an odd ring) and then invokes a `Compute MSB` subroutine to compute the most significant bit (MSB) which is closely related to the DReLU function. Note that DReLU, when using fixed point encoding over a ring $\mathbb{Z}_L$ is defined as follows:

$$\text{DReLU}(x) = \begin{cases} 0 & \text{if } x > L/2 \\ 1 & \text{Otherwise} \end{cases} \tag{1}$$

Each of these subroutines have roughly the same overhead. In Falcon, we show an easier technique using new mathematical insights to compute DReLU which reduces the overhead by over 2×. Note that ReLU and DReLU, non-linear activation functions central to deep learning, are typically the expensive operations in MPC. The first two points above lead to strictly improved protocol for these. Third, Falcon uses a smaller ring size while using an exact yet expensive truncation protocol. This trade-off however allows the entire framework to operate on smaller data-types, thus reducing the communication complexity at least 2×. Furthermore, this communication improvement is amplified with the superlinear dependence of the overall communication on the ring size (cf Table 9). This reduced communication is critical to the communication improvements of Falcon over prior work. In other words, we notice strictly larger performance improvements (than the theoretical improvements) in our end-to-end deployments of benchmarked networks presented in Section 5.

Improved Scope of ML Algorithms. Prior works focus on implementing protocols for linear layers and important non-linear operations. We propose and implement an end-to-end protocol for batch normalization (both forward and backward pass). Batch-normalization is widely used in practice for speedy training of neural networks and is critical for machine learning for two reasons. First, it speeds up training by allowing higher learning rates and prevents extreme values of activations [39]. This is an important component of the parameter tuning for neural networks as there is limited "seeing and learning" during private training. Second, it reduces over-fitting by providing a slight regularization effect and thus improves the stability of training [39]. In other words, private training of neural networks without batch normalization is generally difficult and requires significant pre-training. To truly enable private deep learning, efficient protocols for batch-normalization are required. Implementing batch normalization in MPC is hard for two reasons, first computing the inverse of a number is generally difficult in MPC. Second, most approximate approaches require the inputs to be within a certain range, i.e., there is a trade-off between having an approximate function for inverse of a number over a large range and the complexity of implementing it in MPC. Through our implementation, we enable batch normalization that can allow the training of complex network architectures such as AlexNet (about 60 million parameters).

2.3.1 Comprehensive Evaluation

As shown in Table 1, there are a number of factors involved in comparing different MPC protocols and that none of the prior works provide a holistic solution. We also thoroughly benchmark our proposed system – we evaluate our approach over 6 different network architectures and over 3 standard datasets (MNIST, CIFAR-10, and Tiny ImageNet). We also benchmark our system in both the LAN and WAN setting, for training as well as for inference, and in both the semi-honest and actively secure adversarial models. Finally, we provide a thorough performance comparison against prior state-of-the-art works in the space of privacy preserving machine learning (including 2PC, purely for the sake of comprehensive comparison). We believe that such a comparison, across a spectrum of deployment scenarios, is useful for MPC practitioners.

Finally, we note that the insights and techniques developed in this work are broadly applicable. For instance, ReLU is essentially a comparison function which can thus enable a number of other applications – private computation of decision trees, privacy-preserving searching and thresholding, and private sorting.

3 Protocol Constructions

We begin by describing the notation used in this paper. We then describe how basic operations are performed over the secret sharing scheme and then move on to describe our protocols in detail.

3.1 Notation

Let P_1, P_2, P_3 be the parties. We use P_{i+1}, P_{i-1} to denote the next and previous party for P_i (with periodic boundary conditions). In other words, next party for P_3 is P_1 and previous party for P_1 is P_3. We use $[\![x]\!]^m$ to denote 2-out-of-3 replicated secret sharing (RSS) modulo m for a general modulus m. For any x let $[\![x]\!]^m = (x_1, x_2, x_3)$ denote the RSS of a secret x modulo m i.e., $x \equiv x_1 + x_2 + x_3 \pmod{m}$, but they are otherwise random. We use the notation $[\![x]\!]^m$ to mean (x_1, x_2) is held by P_1, (x_2, x_3) by P_2, and (x_3, x_1) by P_3. We denote by $x[i]$ the i^{th} component of a vector x. In this work, we focus on three different moduli $L = 2^\ell$, a small prime p, and 2. In particular, we use $\ell = 2^5$, $p = 37$. We use fixed-point encoding with 13 bits of precision. In Π_{Mult} over $\mathbb{Z}_p$, the multiplications are performed using the same procedure with no truncation. ReLU, which compares a value with 0 in this representation corresponds to a comparison with $2^{\ell-1}$.

3.2 Basic Operations

To ease the exposition of the protocols, we first describe how basic operations can be performed over the above secret sharing scheme. These operations are extensions of Boolean computations from Araki *et al.* [30] to arithmetic shares, similar to ABY^3 [11]. However, ABY^3 relies on efficient garbled circuits for non-linear function computation which is fundamentally different than the philosophy of this work which relies on simple modular arithmetic. In this manner, we propose a hybrid integration of ideas from SecureNN and ABY^3.

Correlated Randomness: Throughout this work, we will need two basic random number generators. Both of these can be efficiently implemented (using local computation) using PRFs. We describe them below:

- *3-out-of-3 randomness:* Random $\alpha_1, \alpha_2, \alpha_3$ such that $\alpha_1 + \alpha_2 + \alpha_3 \equiv 0 \pmod{L}$ and party P_i holds α_i.
- *2-out-of-3 randomness:* Random $\alpha_1, \alpha_2, \alpha_3$ such that $\alpha_1 + \alpha_2 + \alpha_3 \equiv 0 \pmod{L}$ and party P_i holds (α_i, α_{i+1}).

Given pairwise shared random keys k_i (shared between parties P_i and P_{i+1}), the above two can be computed as $\alpha_i = F_{k_i}(\mathsf{cnt}) - F_{k_{i-1}}(\mathsf{cnt})$ and $(\alpha_i, \alpha_{i-1}) = (F_{k_i}(\mathsf{cnt}), F_{k_{i-1}}(\mathsf{cnt}))$ where cnt is a counter incremented after each invocation. This is more formally described later on in Π_{Prep} in Fig. 6.

Linear Operations: Let a, b, c be public constants and $[\![x]\!]^m$ and $[\![y]\!]^m$ be secret shared. Then $[\![ax+by+c]\!]^m$ can be locally computed as $(ax_1+by_1+c, ax_2+by_2, ax_3+by_3)$ and hence are simply local computations.

Multiplications Π_{Mult}: To multiply two shared values together $[\![x]\!]^m = (x_1, x_2, x_3)$ and $[\![y]\!]^m = (y_1, y_2, y_3)$, parties locally compute $z_1 = x_1y_1 + x_2y_1 + x_1y_2$, $z_2 = x_2y_2 + x_3y_2 + x_2y_3$ and $z_3 = x_3y_3 + x_1y_3 + x_3y_1$. At the end of this, z_1, z_2 and z_3 form a 3-out-of-3 secret sharing of $[\![z = x \cdot y]\!]^m$. Parties then perform *resharing* where 3-out-of-3 randomness is used to generate 2-out-of-3 sharing by sending $\alpha_i + z_i$ to party $i-1$ where $\{\alpha_i\}$ form a 3-out-of-3 secret sharing of 0.

Convolutions and Matrix Multiplications: We rely on prior work to perform convolutions and matrix multiplications over secret shares. To perform matrix multiplications, we note that Π_{Mult} described above extends to incorporate matrix multiplications. To perform convolutions, we simply expand the convolutions into matrix multiplications of larger dimensions (cf Section 5.1 of [12]) and invoke the protocol for matrix multiplications. With fixed-point arithmetic, each multiplication protocol has to be followed by the truncation protocol (cf Fig. 6) to ensure correct fixed-point precision. For more details on fixed-point multiplication, semi-honest, and malicious variants of this refer to [11, 21].

Reconstruction of $[\![x]\!]^m$: In the semi-honest setting, each party sends one ring element to the next party, i.e., P_i sends share x_i to P_{i+1}. In the malicious setting, each party sends x_i to P_{i+1} and x_{i+1} to P_{i-1} and aborts if the two received values do not agree. In either case, a single round of communication is required.

Select Shares Π_{SS}: We define a sub-routine Π_{SS}, which will be used a number of times in the descriptions of other functionalities. It takes as input shares of two random values $[\![x]\!]^L$, $[\![y]\!]^L$, and shares of a random bit $[\![b]\!]^2$. The output $[\![z]\!]^L$ is either $[\![x]\!]^L$ or $[\![y]\!]^L$ depending on whether $b = 0$ or $b = 1$. To do this, we assume access to shares of a random bit $[\![c]\!]^2$ and $[\![c]\!]^L$ (pre-computation). Then we open the bit $(b \oplus c) = e$. If $e = 1$, we set $[\![d]\!]^L = [\![1-c]\!]^L$ otherwise set $[\![d]\!]^L = [\![c]\!]^L$. Finally, we compute $[\![z]\!]^L = [\![(y-x) \cdot d]\!]^L + [\![x]\!]^L$ where $[\![(y-x) \cdot d]\!]^L$ can be computed using Π_{Mult} $(y-x, d)$.

XOR with Public Bit b: Given shares of a bit $[\![x]\!]^m$ and a public bit b, we can locally compute shares of bit $[\![y]\!]^m = [\![x \oplus b]\!]^m$ by noting that $y = x + b - 2b \cdot x$. Since b is public, this is a linear operation and can be computed in both the semi-honest and malicious adversary models.

Evaluating $[\![(-1)^\beta \cdot x]\!]^m$ from $[\![x]\!]^m$ and $[\![\beta]\!]^m$: We assume that $\beta \in \{0, 1\}$. We first compute $[\![1-2\beta]\!]^m$ and then perform the multiplication protocol described above to obtain $[\![(1-2\beta) \cdot x]\!]^m = [\![(-1)^\beta \cdot x]\!]^m$. We split our computations into data dependent online computations and data independent offline computations. Protocols for offline computations are presented in Fig. 6.

3.3 Private Compare

This function evaluates the bit $x \geq r$ where r is public and the parties hold shares of bits of x in $\mathbb{Z}_p$. Algorithm 1 describes this protocol. Note that β is necessary for privacy as β' reveals information about the output $(x \geq r)$ if not blinded by a random bit β. Each of the bits are independent so a single blinding bit β is sufficient to hide computation of $(x \geq r)$ or $(r > x)$.

(A) Step 2: $u[i]$ can be computed by first evaluating shares of $2\beta - 1$ and then computing the product of $(2\beta - 1)$ and $x[i] - r[i]$. This can be done in a single round using one invocation of Π_{Mult}.

(B) Steps 3,4: These are simply local computations. For instance, $[\![w[i]]\!] = (w[i]_1, w[i]_2, w[i]_3)$ can be computed as $w[i]_j = x[i]_j + \delta_{j1}r[i] - 2r[i]x[i]_j$ where $j \in \{1,2,3\}$ and δ_{ij} is the Kronecker delta function and is 1 if $i = j$ and 0 otherwise.

(C) Step 6 can be computed in $\log_2 \ell + 1$ rounds using sequential invocations of the Π_{Mult} with smaller strings (One additional round because of multiplication by random blinding factor).

(D) Steps 7,8: These are once again local computations.

This protocol is an example of the challenges of integrating approaches based on simple modular arithmetic with malicious security. Both SecureNN and FALCON aim to find if there exists an index i such that $c[i] = 0$. However, the existence of a semi-honest third party makes checking this much easier in SecureNN. The two primary parties simply blind and mask their inputs and send them to the third party. This is not possible in FALCON due to the stronger adversarial model and requires newer protocol constructions. In particular, we need to multiply all the $c[i]$'s together along with a mask in $\mathbb{Z}_p^*$ and reveal this final product to compute the answer.

Algorithm 1 Private Compare $\Pi_{\mathsf{PC}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold secret sharing of bits of x in $\mathbb{Z}_p$.
Output: All parties get shares of the bit $(x \geq r) \in \mathbb{Z}_2$.
Common Randomness: P_1, P_2, P_3 hold a public ℓ bit integer r, shares of a random bit in two rings $[\![\beta]\!]^2$ and $[\![\beta]\!]^p$ and shares of a random, secret integer $m \in \mathbb{Z}_p^*$.

```
1: for $i = \{\ell-1, \ell-2, \ldots, 0\}$ do
2:     Compute shares of $u[i] = (-1)^\beta (x[i] - r[i])$
3:     Compute shares of $w[i] = x[i] \oplus r[i]$
4:     Compute shares of $c[i] = u[i] + 1 + \sum_{k=i+1}^{\ell} w[k]$
5: end for
6: Compute and reveal $d := [\![m]\!]^p \cdot \prod_{i=0}^{\ell-1} c[i] \pmod p$
7: Let $\beta' = 1$ if $(d \neq 0)$ and 0 otherwise.
8: return Shares of $\beta' \oplus \beta \in \mathbb{Z}_2$
```

3.4 Wrap Function

Central to the computation of operations such as ReLU and DReLU is a comparison function. The wrap functions, wrap_2 and wrap_3 are defined below as a function of the secret shares of the parties and effectively compute the "carry bit" when the shares are added together as integers. Eq. 11 shows that DReLU can be easily computed using the wrap_3 function. So all we require is a secure protocol for wrap_3. We define two similar functions called "wrap" (denoted by wrap_2 and wrap_3). We call Eq. 3 the *exact wrap* function. For the remainder of the paper, we use the (mod 2) reduction of the wrap function defined in Equation 4 and refer to it as simply the wrap function.

$$\mathsf{wrap}_2(a_1, a_2, L) = \begin{cases} 0 & \text{if } a_1 + a_2 < L \\ 1 & \text{Otherwise} \end{cases} \tag{2}$$

$$\mathsf{wrap}_{3e}(a_1, a_2, a_3, L) = \begin{cases} 0 & \text{if } \sum_{i=1}^{3} a_i < L \\ 1 & \text{if } L \leq \sum_{i=1}^{3} a_i < 2L \\ 2 & \text{if } 2L \leq \sum_{i=1}^{3} a_i < 3L \end{cases} \tag{3}$$

$$\mathsf{wrap}_3(a_1, a_2, a_3, L) = \mathsf{wrap}_{3e}(a_1, a_2, a_3, L) \pmod 2 \tag{4}$$

Next we briefly describe the connection between wrap_3 computed on shares a_1, a_2, a_3 and the most significant bit (MSB) of the underlying secret a. Note that $a = a_1 + a_2 + a_3 \pmod L$ as a_i's are shares of a modulo L. Considering this sum as a logic circuit (for instance as a ripple carry adder), we can see that $\text{MSB}(a) = \text{MSB}(a_1) + \text{MSB}(a_2) + \text{MSB}(a_3) + c \pmod 2$ where c is the carry bit from the previous index. The key insight here is that the carry c from the previous index is simply the wrap_3 function computed on a_i's (ignoring their MSB's) modulo $L/2$ (this is evident from Eq. 3). And this last operation is synonymous with computing the wrap_3 function on $2a_i$'s modulo L. We will further describe the consequences of this connection in Section 3.5 where we describe a protocol to compute the ReLU and DReLU functions. Algorithm 2 gives the protocol for securely computing the wrap_3 function.

Note that wrap_2 function is always computed locally and hence a secure algorithm is not needed for the same. Furthermore, note that the wrap_2 function allows us to write exact integer equations as follows: if $a \equiv a_1 + a_2 \pmod L$ then $a = a_1 + a_2 - \mathsf{wrap}_2(a_1, a_2, L) \cdot L$ where the former relation is a congruence relation but the latter is an integer relation (and has exact equality). Finally, to see the correctness of the wrap_3 protocol, in reference to

Algorithm 2 wrap$_3$ $\Pi_{\mathsf{WA}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a bit $\theta = \mathsf{wrap}_3(a_1, a_2, a_3, L)$
Common Randomness: P_1, P_2, P_3 hold shares $[\![x]\!]^L$ (of a random number x), $[\![x[i]]\!]^p$ (shares of bits of x) and $[\![\alpha]\!]^2$ where $\alpha = \mathsf{wrap}_3(x_1, x_2, x_3, L)$.

```
1: Compute $r_j \equiv a_j + x_j \pmod{L}$ and $\beta_j = \mathsf{wrap}_2(a_j, x_j, L)$
2: Reconstruct $r \equiv \sum r_j \pmod{L}$
3: Compute $\delta = \mathsf{wrap}_3(r_1, r_2, r_3, L)$        ▷ In the clear
4: Run $\Pi_{\mathsf{PC}}$ on $x, r+1$ to get $\eta = (x \geq r+1)$.
5: return $\theta = \beta_1 + \beta_2 + \beta_3 + \delta - \eta - \alpha$
```

Algorithm 2, we can write the following set of equations

$$r = a + x - \eta \cdot L \tag{5}$$

$$r = r_1 + r_2 + r_3 - \delta_e \cdot L \tag{6}$$

$$r_i = a_i + x_i - \beta_i \cdot L \qquad \forall i \in \{1, 2, 3\} \tag{7}$$

$$x = x_1 + x_2 + x_3 - \alpha_e \cdot L \tag{8}$$

where δ_e, α_e denote the exact wrap functions, Eq. 6,8 follow from the definition of the exact wrap function, while Eq 7 follows from the definition of wrap_2 function. To see Eq. 5, note that $r, a, x \in [0, L-1]$ and that $r \equiv a + x \pmod{L}$. Hence $a + x \geq L$ iff $r < x$ (or $x \geq r + 1$). Finally, assuming θ_e is the exact wrap function on a_1, a_2, a_3 i.e.,

$$a = a_1 + a_2 + a_3 - \theta_e \cdot L \tag{9}$$

Eqs. 5-9 together give a constraint among the Greek symbols (in other words, (5) - (6) - (7) + (8) + (9) gives Eq. 10 below)

$$\theta_e = \beta_1 + \beta_2 + \beta_3 + \delta_e - \eta - \alpha_e \tag{10}$$

Reducing Eq. 10 modulo 2 gives us $\theta = \beta_1 + \beta_2 + \beta_3 + \delta - \eta - \alpha$ which is used to compute wrap_3 as in Algorithm 2.

3.5 ReLU and Derivative of ReLU

We now describe how to construct a protocol for securely computing ReLU(a) and DReLU(a) for a given secret a. Recall that we use fixed point arithmetic over $\mathbb{Z}_{2^\ell}$ for efficiency reasons. Using the natural encoding of native `C++` data-types, we know that positive numbers are the first $2^{\ell-1}$ and have their most significant bit equal to 0. Negative numbers, on the other hand are the last $2^{\ell-1}$ numbers in the ℓ-bit range and have their most significant bit equal to 1. Thus, the DReLU function defined by Eq. 1, has a simple connection with the most significant bit (MSB) of the fixed point representation viz., DReLU(a) = 1 − MSB(a). Furthermore, in Section 3.4, we have seen the connection between MSB(a) and wrap_3. Together, these insights can be distilled into the following equation:

$$\begin{aligned}\mathsf{DReLU}(\mathsf{a}) = \mathsf{MSB}(a_1) \oplus \mathsf{MSB}(a_2) \oplus \mathsf{MSB}(a_3) \\ \oplus\, \mathsf{wrap}_3(2a_1, 2a_2, 2a_3, L) \oplus 1\end{aligned} \tag{11}$$

In particular, Derivative of ReLU can be computed by combining the output of the wrap function with local computations. Finally, for computing ReLU from DReLU, we simply call Π_{SS} (which effectively performs Π_{Mult} on shares of a and shares of $\mathsf{DReLU}(a)$). With these observations, we can implement the ReLU and Derivative of ReLU protocols (see Algorithm 3). Note that the approach here is crucially different from the approach SecureNN uses due to use of fundamentally different building blocks as well as deeper mathematical insights such as Eq. 11. To achieve the DReLU functionality, SecureNN first uses a subroutine to transform the shares of the secret into an odd modulus ring and then uses another subroutine to compute the MSB (cf Section 2.3). Both these subroutines have similar complexities. FALCON on the other hand uses the insight presented in Eq. 11 to completely eliminate the need for these subroutines, improving the efficiency by about 2× and simplifying the overall protocol. This also drastically improves the end-to-end performance (by over 6.4×) as the ReLU and DReLU functionalities are the building blocks of every comparison in the network.

Algorithm 3 ReLU, $\Pi_{\mathsf{ReLU}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of $\mathsf{ReLU}(a)$.
Common Randomness: $[\![c]\!]^2$ and $[\![c]\!]^L$ (shares of a random bit in two rings)

```
1: Run $\Pi_{\mathsf{WA}}$ to get $\mathsf{wrap}_3(2a_1, 2a_2, 2a_3, L)$
2: Compute $[\![b]\!]^2$ where $b = \mathsf{DReLU}(a)$  ▷ Local comp. (Eq. 11)
3: return Output of $\Pi_{\mathsf{SS}}$ run on $\{a, 0\}$ with $b$ as selection.
```

3.6 Maxpool and Derivative of Maxpool

The functionality of maxpool simply takes as input a vector of secret shared values and outputs the maximum value. For derivative of maxpool, we need a one-hot vector of the same size as the input where the 1 is at the location of the index of the maximum value. Maxpool can be implemented using a binary sort on the vector of inputs and small amounts of bookkeeping, where the comparisons can be performed using ReLUs. Derivative of maxpool can be efficiently implemented along with maxpool described in Algorithm 4.

Algorithm 4 Maxpool, $\Pi_{\mathsf{Maxpool}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of $a_1, a_2, \ldots a_n$ in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a_k and $\boldsymbol{e_k}$ where $k = \mathsf{argmax}\{a_1, \ldots a_n\}$ and where $\boldsymbol{e_k} = \{e_1, e_2, \ldots e_n\}$ with $e_i = 0\ \forall i \neq k$ and $e_k = 1$.
Common Randomness: No additional common randomness required.

```
1: Set $\mathsf{max} \leftarrow a_1$ and $\mathbf{ind} \leftarrow \boldsymbol{e_1} = \{1, 0, \ldots, 0\}$
2: for $i = \{2, 3, \ldots n\}$ do
3:     Set $\mathsf{d_{max}} \leftarrow (\mathsf{max} - a_i)$ and $\mathbf{d_{ind}} \leftarrow (\mathbf{ind} - \boldsymbol{e_i})$
4:     $b \leftarrow \Pi_{\mathsf{DReLU}}(\mathsf{d_{max}})$        ▷ b → Derivative of ReLU
5:     $\mathsf{max} \leftarrow \Pi_{\mathsf{SS}}$ on inputs $\{a_i, \mathsf{max}, b\}$.
6:     $\mathbf{ind} \leftarrow \Pi_{\mathsf{SS}}$ on inputs $\{\boldsymbol{e_i}, \mathbf{ind}, b\}$.
7: end for
8: return $\mathsf{max}, \mathbf{ind}$
```

Algorithm 5 Bounding Power, $\Pi_{\mathsf{Pow}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of x in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get α in the clear, where $2^\alpha \leq x < 2^{\alpha+1}$.
Common Randomness: No additional common randomness required.

```
1: Initialize $\alpha \leftarrow 0$
2: for $i = \{\ell - 1, \ldots, 1, 0\}$ do
3:     $c \leftarrow \Pi_{\mathsf{DReLU}}(\mathsf{x} - 2^{2^i + \alpha})$ and reconstruct $c$
4:     Set $\alpha \leftarrow \alpha + 2^i$ if $c = 1$
5: end for
6: return $\alpha$
```

3.7 Division and Batch Normalization

Truncation allows parties to securely eliminate lower bits of a secret shared value (i.e., truncation by k bits of a secret $a \rightarrow a/2^k$). However, the problem of dividing by a secret shared number is considerably harder and efficient algorithms rely on either (1) sequential comparison (2) numerical methods. In this work, we use the numerical methods approach for its efficiency. We use the specific choices of initializations given in [40, 41] to efficiently compute division over secret shares. A crucial component of numerical methods is the need to estimate the value of the secret within a range. We achieve this using Algorithm 5. Note that Algorithm 5 outputs the bounding power of 2, which is also what is guaranteed by the functionality. In this way, we only reveal the bounding power of 2 and nothing else.

Algorithm 6 is used to compute the value of a/b where a, b are secret shared. The first step for the algorithm is to transform $b \rightarrow x$ where $x \in [0.5, 1)$. Note that even though b is a fixed-point precision of f_p, for the computations in Algorithm 6, x has to be interpreted as a value with fixed-point precision $\alpha + 1$ where $2^\alpha \leq b < 2^{\alpha+1}$. Thus we first need to extract α (the appropriate range) using Algorithm 5. Let $w_0 = 2.9142 - 2x$, $\epsilon_0 = 1 - x \cdot w_0$ (cf. [40, 41] for choice of constants). Then an initial approximation for $1/x$ is $w_0 \cdot (1 + \epsilon_0)$. For higher order approximations, set $\epsilon_i = \epsilon_{i-1}^2$ and multiply the previous approximate result by $(1 + \epsilon_i)$ to get a better approximate result. Each successive iteration increases the round complexity by 2. For our value of fixed-point precision, we use the following approximation which works with high accuracy (refer to Section 5 for details):

Algorithm 6 Division, $\Pi_{\mathsf{Div}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a, b in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a/b in $\mathbb{Z}_L$ computed as integer division with a given fixed precision f_p.
Common Randomness: No additional common randomness required.

```
1: Run $\Pi_{\mathsf{Pow}}$ on $b$ to get $\alpha$ such that $2^\alpha \leq b < 2^{\alpha+1}$
2: Compute $w_0 \leftarrow 2.9142 - 2b$
3: Compute $\epsilon_0 \leftarrow 1 - b \cdot w_0$ and $\epsilon_1 \leftarrow \epsilon_0^2$
4: return $a w_0 (1 + \epsilon_0)(1 + \epsilon_1)$
```

Algorithm 7 Batch Norm, $\Pi_{\mathsf{BN}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of $a_1, a_2 \ldots a_m$ in $\mathbb{Z}_L$ where m is the size of each batch and shares of two learnable parameters γ, β.
Output: P_1, P_2, P_3 get shares of $\gamma z_i + \beta$ for $i \in [m]$ and $z_i = (a_i - \mu)/(\sqrt{\sigma^2 + \epsilon})$ where $\mu = 1/m \sum a_i$, $\sigma^2 = 1/m \sum (a_i - \mu)^2$, and ϵ is a set constant.
Common Randomness: No additional common randomness required.

```
1: Set $\mu \leftarrow 1/m \cdot \sum a_i$
2: Compute $\sigma^2 \leftarrow 1/m \cdot \sum (a_i - \mu)^2$ and let $b = \sigma^2 + \epsilon$
3: Run $\Pi_{\mathsf{Pow}}$ on $b$ to find $\alpha$ such that $2^\alpha \leq b < 2^{\alpha+1}$
4: Set $x_0 \leftarrow 2^{-\lfloor \alpha/2 \rceil}$
5: for $i \in 0, \ldots, 3$ do
6:     Set $x_{i+1} \leftarrow \frac{x_i}{2}(3 - b x_i^2)$
7: end for
8: return $\gamma \cdot x_{\mathsf{rnds}} \cdot (a_i - \mu) + \beta$ for $i \in [m]$
```

$$\mathsf{AppDiv}(x) = w_0 \cdot (1 + \epsilon_0)(1 + \epsilon_1) \approx \frac{1}{x} \tag{12}$$

Batch-norm is another important component of neural network architectures. They improve the convergence as well as help automate the training process. Algorithm 7 describes the protocol to compute batch-norm. For step 3, we use Newton's method and use $2^{-\lfloor \alpha/2 \rceil}$ as an initial approximation of $1/\sqrt{\sigma^2 + \epsilon}$, where $2^\alpha \leq \sigma^2 + \epsilon < 2^{\alpha+1}$ and use the successive iterative formula:

$$x_{n+1} = \frac{x_n}{2}\left(3 - a x_n^2\right) \tag{13}$$

Given the choice of initial guess, 4 rounds are sufficient for a close approximation with our choice of fixed-point precision. However, batch normalization during training is computed by sequentially computing $\sqrt{\sigma^2+\epsilon}$ and then computing the inverse. This approach is used to optimize the computation required during back-propagation which requires the values of $\sqrt{\sigma^2+\epsilon}$. For computing the square root of a value a, we use Newton's method given by Eq. 14. This can then be used in conjunction with the inverse computation given by Eq. 12 to complete the batch-norm computations.

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{14}$$

4 Theoretical Analysis

We provide theoretical analysis of our framework and protocols. In particular, we provide proofs of security and analyze the theoretical complexity.

4.1 Security Proofs

We model and prove the security of our construction in the real world-ideal world simulation paradigm [42–44]. In the real interaction, the parties execute the protocol in the presence of an adversary and the environment. On the other hand, in the ideal interaction, the parties send their inputs to a trusted party that computes the functionality truthfully. Finally, to prove the security of our protocols, for every adversary in the real interaction, there exists a simulator in the ideal interaction such that the environment cannot distinguish between the two scenarios. In other words, whatever information the adversary extracts in the real interaction, the simulator can extract it in the ideal world as well.

We show that our protocols are perfectly secure (i.e., the joint distributions of the inputs, outputs, and the communication transcripts are exactly the same and not statistically close) in the stand-alone model (i.e., protocol is executed only once), and that they have a straight-line black-box simulators (i.e., only assume oracle access to the adversary and hence do no rewind). We then rely on the result of Kushilevitz *et al.* [38] to prove that our protocols are secure under concurrent general composition (Theorem 1.2 in [38]).

Due to space constraints, we formally describe the functionalities in Appendix E. We describe simulators for Π_{PC} (Fig. 3), Π_{WA} (Fig. 4), Π_{ReLU} (Fig. 5), Π_{Maxpool} (Fig. 7), Π_{Pow} (Fig. 8), Π_{Div} (Fig. 9), and Π_{BN} (Fig. 10) that achieve indistinguishability. $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Trunc}}, \mathcal{F}_{\mathsf{Reconst}}$ are identical to prior works [11, 21]. We prove security using the standard indistinguishability argument. To prove the security of a particular functionality, we set-up hybrid interactions where the sub-protocols used in that protocol are replaced by their corresponding ideal functionalities and then prove that the interactions can be simulated. This hybrid argument in effect sets up a series of interactions $I_0, I_1, \ldots I_k$ for some k where I_0 corresponds to the real interaction and I_k corresponds to the ideal interaction. Each neighboring interaction, i.e., I_i, I_{i+1} for $i \in \{0, \ldots, k-1\}$ is then shown indistinguishable from each other, in effect showing that the real and ideal interactions are indistinguishable. Without loss of generality, we assume that party P_2 is corrupt. In the *real world*, the adversary $\mathcal{A}$ interacts with the honest parties P_0 and P_1. In the *ideal world*, the simulator interacts with the adversary and simulates exact transcripts for interactions between the adversary $\mathcal{A}$ and P_0, P_1. On the other hand, the simulator should be able to extract the adversaries inputs (using the values for the inputs of the honest parties in the internal run and the fact that each honest party has one component of the replicated secret sharing). These inputs are fed to the functionality to generate correct output distributions, thus achieving security against malicious adversaries. Theorems 1-6 gives the indistinguishability of these two interactions.

Theorem 1. Π_{PC} *securely realizes* $\mathcal{F}_{\mathsf{PC}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. We first set up some detail on the proof strategy that is essential for other proofs as well. For the ease of exposition, we describe it in the context of Π_{PC}. The goal of designing a simulator is to be able to demonstrate the ability to produce transcripts that are indistinguishable from the transcripts in the real world. The joint distribution of the inputs and outputs is a part of these transcripts and hence has to be indistinguishable in the two interactions. However, since the honest parties simply forward their inputs to the functionality, the simulator must be able to extract the inputs of the malicious parties to be able to generate the correct shares for the honest parties.

The usual technique to achieve this is to have the simulator run a simulated version of the protocol internally, i.e., emulating the roles of the honest parties and interacting with the adversary. This is what we call an *internal run*. This internal run can then be used to extract the inputs of the adversarial party (which can then be forwarded to the functionality in the ideal interaction). In the hybrid argument, the subroutines used in the protocol can be replaced by their corresponding ideal interactions, the simulator can emulate the roles of these trusted functionalities in its internal run.

In the specific context of Π_{PC}, the simulator $\mathcal{S}$ for adversary $\mathcal{A}$ works by playing the role of the trusted party for $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Reconst}}$ and $\mathcal{F}_{\mathsf{Prep}}$. To be able to simulate, we need to show that:

(1) Real interaction transcripts can be simulated.
(2) Honest parties receive their outputs correctly.

Simulation follows easily from the protocol and the hybrid argument. The simulator for Π_{Mult} (along with the simulator for Π_{Reconst}) can be used to simulate the transcripts from Steps 2, 6 (from Algorithm 1). Note that the distributions of these transcripts are all uniformly random values (β is required to make the transcript for β' uniformly random, the various bits $u[i], w[i]$, and $c[i]$ are random because x is random) and hence achieve perfect security. Steps 3, 4, 7, and 8 on the other hand are all local and do not need simulation.

To extract the inputs of the malicious party, the simulator uses the fact that it has access to r and β (though $\mathcal{F}_{\mathsf{Prep}}$) and all the internal values for the honest parties (in the internal run) and hence can extract the shares of $x[i]$ from the corrupt party P_2. Finally, if the protocol aborts at any time in the internal run, then the simulator sends Abort to $\mathcal{F}_{\mathsf{PC}}$ otherwise, it inputs the extracted shares of $x[i]$ to $\mathcal{F}_{\mathsf{PC}}$ and the honest parties receive their outputs. □

Theorem 2. Π_{WA} *securely realizes* $\mathcal{F}_{\mathsf{WA}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{PC}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. We use a similar set-up as the proof of Theorem 1. Step 1 is local computation and does not need simulation. Steps 2, 4 can be simulated using the simulators for $\mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{PC}}$ respectively. Input extraction follows from having access to r_i (through $\mathcal{F}_{\mathsf{Prep}}$) and output x if the protocol does not abort. If the protocol does abort at any time in the internal run, then the simulator sends Abort to $\mathcal{F}_{\mathsf{WA}}$. Otherwise, it simply passes on the extracted shares of $a[i]$ to $\mathcal{F}_{\mathsf{WA}}$ and the honest parties receive their outputs. Note that Π_{DReLU} is not formally defined. However, this is simply local computation over Π_{WA} and the proofs can be extended analogously. □

Theorem 3. Π_{ReLU} *securely realizes* $\mathcal{F}_{\mathsf{ReLU}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{WA}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Simulation is done using the hybrid argument. The protocol simply composes $\mathcal{F}_{\mathsf{WA}}$ and $\mathcal{F}_{\mathsf{Mult}}$ and hence is simulated using the corresponding simulators. □

Theorem 4. Π_{Maxpool} *securely realizes* $\mathcal{F}_{\mathsf{Maxpool}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Similar to the proof of Theorem 3, simulation works by sequentially composing the simulators for $\mathcal{F}_{\mathsf{ReLU}}$ and $\mathcal{F}_{\mathsf{Mult}}$. □

Theorem 5. Π_{Pow} *securely realizes* $\mathcal{F}_{\mathsf{Pow}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. The simulator for $\mathcal{A}$ works by playing the role of the trusted party for $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}$, and $\mathcal{F}_{\mathsf{Reconst}}$. The protocol sequentially reveals bits of the scale α. It is important to note the functionality that it emulates (see in Fig. 8). The simulator runs the first iteration of the loop and in the process extracts the adversaries inputs. Then it proceeds to complete all the iterations of the loop. If the protocol proceeds without aborting till the end, then the simulator sends the extracted shares of b along with $k = 0$ to the functionality $\mathcal{F}_{\mathsf{Pow}}$. If the protocol aborts at iteration k, then the simulator sends the extracted shares of b along with k to $\mathcal{F}_{\mathsf{Pow}}$. □

Theorem 6. Π_{Div}, Π_{BN} *securely realize* $\mathcal{F}_{\mathsf{Div}}$, $\mathcal{F}_{\mathsf{BN}}$ *respectively, with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Pow}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Π_{Div}, Π_{BN} are sequential combinations of local computations and invocations of $\mathcal{F}_{\mathsf{Mult}}$. Simulation follows directly from composing the simulators and input extraction follows from the simulator of Π_{Pow}. □

4.1.1 Protocol Overheads

We theoretically estimate the overheads of our protocols in Table 9 in Appendix B. The dominant round complexity for private compare comes from the string multiplication in Step 6. wrap_3 requires one additional round and one additional ring element (two in malicious security) over private compare. Computing derivative of ReLU is a local computation over the wrap_3 function. Computing ReLU requires two additional rounds and one ring element (two for malicious). Maxpool and derivative of require rounds proportional to the area of the filter. Finally, pow, division, and batch-norm requires a quadratic number of rounds in ℓ.

5 Experimental Evaluation

We evaluate the performance of training and inference with FALCON on 6 networks of varying parameter sizes trained using MNIST, CIFAR-10 and Tiny ImageNet datasets (cf. Appendix C). A number of prior works such as SecureML [4], DeepSecure [45], MiniONN [6], Gazelle [8], SecureNN [12], ABY^3 [11], and Chameleon [5] evaluate over these networks and we mimic their evaluation set-up for comparison.

5.1 Experimental Setup

We implement Falcon framework in about 14.6k LOC in C++ using the communication backend of SecureNN and will be open-sourced at https://github.com/snwagh/falcon-public. We run our experiments on Amazon EC2 machines over Ubuntu 18.04 LTS with Intel-Core i7 processor and 64GB of RAM. Our evaluation set-up uses similar as compared to prior work [4, 5, 8, 11, 12]. We perform extensive evaluation of our framework in both the LAN and WAN setting. For the LAN setting, our bandwidth is about 625 MBps and ping time is about 0.2ms. For WAN experiments, we run servers in different geographic regions with 70ms ping time and 40 MBps bandwidth.

Optimizations: All data-independent computation, i.e., pre-computation, is parallelized using 16 cores to reduce the run-time. When a ReLU layer is followed by a Maxpool layer, we swap the order of these two layers for optimized runtimes. We use the Eigen library for faster matrix multiplication and parallelize the private compare computation. We optimize across the forward and backward pass for Maxpool, ReLU, and Batch-Normalization layers, i.e., we compute the relevant derivatives while computing the functions. We use 32-bit integer range with 16 bits of fixed-point precision. As the entire codebase is parallelizable, significant improvement is possible by implementing Falcon using TensorFlow or PyTorch which support easy parallelization as well as computations over GPUs.

Networks and Datasets: For comparison with different networks as well as plaintext computations, we select 3 standard benchmarking datasets — MNIST [27], CIFAR-10 [28], and Tiny ImageNet [29] and 6 standard network architectures – 3 from the privacy-preserving ML community and 3 from the ML community. For more details refer to Appendix C, D.

5.2 Results for Private Inference

Tables 2, 3 report the end-to-end latency time (in seconds) and number of bytes (in MB) communicated for performing a single inference query with Falcon. We execute the queries in both LAN and WAN as well as semi-honest and malicious settings and compare with prior work wherever applicable.

Comparison to Prior Work. We compare the inference time of a single query and the communication bytes of Falcon with prior work on networks A, B and C. None of the prior works evaluate the remaining networks and hence we do not compare the performance of Falcon for the networks in Table 3. Depending on the network architecture, our results are between 3×-120× faster than existing work. In particular, we are up to 18× faster than XONN [9] (11× on average) and 32× faster than Gazelle (23× on average), 8× faster than SecureNN (3× on average), and comparable to ABY^3 on small networks. We are also 40× more communication efficient than ABY^3 [11], 200× more communication efficient than SecureNN [12], and over 760× more communication efficient compared to XONN [9]. Note that it is hard to compare frameworks without actually running benchmarks as different protocols scale differently for different architectures. For instance, GC based protocols scale better when run over WAN settings and larger networks change the fraction of the total overhead from linear layers for reasons described in Section 5.4 and thus affect different protocols differently.

Inference Time and Communication with Falcon. For both the adversarial settings, the inference latency for Falcon over LAN is within 25ms for smaller networks (A and B) and around 100ms for Network-C and LeNet. For AlexNet and VGG16, the inference time ranges from 0.5 to 12s depending on the model and the input dataset. The inference time increases with the size of the input image. Hence, queries over Tiny ImageNet are slower than CIFAR-10 for the same model architecture. The inference time over the WAN setting ranges from 1 to 3s for the networks A, B and C and from 3 to 37s for the larger networks. However, we emphasize that the inference time with semi-honest adversarial setting is around 3× faster than that for the malicious adversary. Hence, a faster deployment protocol is possible depending on the trust assumptions of the application.

In addition to efficient response times, our results show that Falcon requires small amounts of communication. Parties exchange less than 4MB of data for smaller networks (Table 2) and 5-400MB for larger networks (Table 3) (same for both LAN, WAN). However, similar to the inference time, malicious setting requires a higher communication and thus higher run-time.

5.3 Results for Private Training

Tables 4, 5 report the execution time and the communication required for training the 6 networks.

	Framework	Threat Model	LAN/ WAN	Network-A		Network-B		Network-C	
				Time	Comm.	Time	Comm.	Time	Comm.
2PC	SecureML [4]	Semi-honest	LAN	4.88	-	-	-	-	-
	DeepSecure [45]	Semi-honest	LAN	-	-	9.67	791	-	-
	EzPC [7]	Semi-honest	LAN	0.7	76	0.6	70	5.1	501
	Gazelle [8]	Semi-honest	LAN	0.09	0.5	0.29	0.8	1.16	70
	MiniONN [6]	Semi-honest	LAN	1.04	15.8	1.28	47.6	9.32	657.5
	XONN [9]	Semi-honest	LAN	0.13	4.29	0.16	38.3	0.15	32.1
3PC	Chameleon [5]	Semi-honest	LAN	-	-	2.7	12.9	-	-
	ABY^3 [11]	Semi-honest	LAN	0.008	0.5	0.01	5.2	-	-
	SecureNN [12]	Semi-honest	LAN	0.043	2.1	0.076	4.05	0.13	8.86
	FALCON	Semi-honest	LAN	0.011	0.012	0.009	0.049	0.042	0.51
		Malicious	LAN	0.021	0.31	0.022	0.52	0.089	3.37
	SecureNN [12]	Semi-honest	WAN	2.43	2.1	3.06	4.05	3.93	8.86
	FALCON	Semi-honest	WAN	0.99	0.012	0.76	0.049	3.0	0.5
		Malicious	WAN	2.33	0.31	1.7	0.52	7.8	3.37
4PC	FLASH [17]	Malicious	LAN	0.029	-	-	-	-	-
	FLASH [17]	Malicious	WAN	12.6	-	-	-	-	-

Table 2. Comparison of inference time of various frameworks for different networks using MNIST dataset. All runtimes are reported in seconds and communication in MB. ABY^3 and XONN do no implement their maliciously secure versions. 2-party (2PC) protocols are presented here solely for the sake of comprehensive evaluation of the literature.

Framework	Threat Model	LAN/WAN	LeNet (MNIST)		AlexNet (CIFAR-10)		VGG16 (CIFAR-10)		AlexNet (ImageNet)		VGG16 (ImageNet)	
			Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
FALCON	Semi-honest	LAN	0.047	0.74	0.043	1.35	0.79	13.51	1.81	19.21	3.15	52.56
	Malicious	LAN	0.12	5.69	0.14	8.85	2.89	90.1	6.7	130.0	12.04*	395.7*
	Semi-honest	WAN	3.06	0.74	0.13	1.35	1.27	13.51	2.43	19.21	4.67	52.56
	Malicious	WAN	7.87	5.69	0.41	8.85	4.7	90.1	8.68	130.0	37.6*	395.7*

Table 3. Comparison of inference time of various frameworks over popular benchmarking network architectures from the machine learning domain. All runtimes are reported in seconds and communication in MB. * indicate non-amortized numbers.

Comparison to Prior Work. For private training, FALCON is up to 6× faster than SecureNN [12] (4× on average), 4.4× faster than ABY^3 and 70× faster than SecureML [4]. We highlight that FALCON achieves these speedups due to improved protocols (both round complexity and communication as described in Section 2.3). Table 4 shows that the communication overhead is 10× to 100× as compared to other solutions.

Execution Time for FALCON. The time to privately train networks A, B and C with FALCON is around 3 to 40 hrs. For larger networks, we extrapolate time from a single iteration of a forward and a backward pass. The training time ranges from a few weeks to hundreds of weeks. Although these values seem to be quite large, high capacity machine learning models are known to take from a few days to weeks to achieve high accuracy when trained (both on CPU as well as GPU). Such networks can also benefit from transfer learning techniques, where a public pre-trained model is fine-tuned with a private dataset. This fine-tuning requires fewer epochs and hence speed up the overall runtime considerably.

5.4 Compute vs. Communication Cost

Figure 1 shows the computation time as compared to the communication time for the inference of a single input over different network sizes. We observe that the computation cost increases with the network size and becomes the dominant reason for the performance overhead in private deep learning with FALCON. The reason for this is because the complexity of matrix multiplication is "super-quadratic" i.e., to multiply two $n \times n$ matrices, the computation overhead is strictly larger than $O(n^2)$. Note that the communication of the matrix multiplication protocol in this work is only linear in the size of the matrices and has a round complexity of a single round. On the other hand, the non-linear operations, though more communication expensive in MPC, are applied on vectors of size equal to the output of the matrix product and thus are "quadratic." In other words, the non-linear operations such as ReLU are applied on the output of the matrix multiplication (FC/Conv layers) and are applied on vectors of size $O(n^2)$ assuming they are applied on the output of the multiplication of two

Framework	Threat Model	LAN/ WAN	Network-A		Network-B		Network-C	
			Time	Comm.	Time	Comm.	Time	Comm.
SecureML [4]*	Semi-honest	LAN	81.7	-	-	-	-	-
SecureML [4]	Semi-honest	LAN	7.02	-	-	-	-	-
ABY^3 [11]	Semi-honest	LAN	0.75	0.031	-	-	-	-
SecureNN [12]	Semi-honest	LAN	1.03	0.11	-	-	17.4	30.6
FALCON	Semi-honest	LAN	0.17	0.016	0.42	0.056	3.71	0.54
	Malicious	LAN	0.56	0.088	1.17	0.32	11.9	3.29
SecureML [4]*	Semi-honest	WAN	4336	-	-	-	-	-
SecureNN [12]	Semi-honest	WAN	7.83	0.11	-	-	53.98	30.6
FALCON	Semi-honest	WAN	3.76	0.016	3.4	56.14	14.8	0.54
	Malicious	WAN	8.01	0.088	7.5	0.32	39.32	3.29
Batch Size, Epochs			128, 15		128, 15		128, 15	

Table 4. Comparison of training time of various frameworks over popular benchmarking network architectures from the security domain. All runtimes are reported in hours and communication in TB. * correspond to 2PC numbers. ABY^3 does not implement their maliciously secure protocols.

Framework	Threat Model	LAN/ WAN	LeNet		AlexNet (CIFAR-10)		VGG16 (CIFAR-10)		AlexNet (ImageNet)		VGG16 (ImageNet)	
			Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
FALCON	Semi-honest	LAN	6.05×10^0	0.81	7.89×10^1	7.24	8.43×10^2	45.9	1.23×10^4	222.9	5.19×10^3	156.0
	Malicious	LAN	1.22×10^1	4.82	2.82×10^2	43.4	3.05×10^3	185.3	4.63×10^4	1598	1.95×10^4	1012
	Semi-honest	WAN	1.85×10^1	0.81	2.33×10^2	7.24	2.09×10^3	45.9	1.54×10^4	222.9	6.89×10^3	156.0
	Malicious	WAN	5.20×10^1	4.82	7.24×10^2	43.4	5.26×10^3	185.3	5.71×10^4	1598	2.47×10^4	1012
Batch Size, Epochs			128, 15		128, 90		128, 25		128, 90		128, 25	

Table 5. Comparison of training time of various frameworks over popular benchmarking network architectures from the machine learning domain. All runtimes are reported in hours and communication in TB.

Network	Training Accuracy	Inference Accuracy	FALCON Inference Accuracy	Relative Error
Network-A	98.18%	97.42%	97.42%	0.471%
Network-B	98.93%	97.81%	97.81%	0.635%
Network-C	99.16%	98.64%	98.64%	0.415%
LeNet	99.76%	99.15%	96.85%	0.965%

Table 6. Summary of experiments involving accuracy of neural networks using secure computation. The first two columns refer to the plaintext accuracies and relative error refers to the average relative error of one forward pass computation using FALCON.

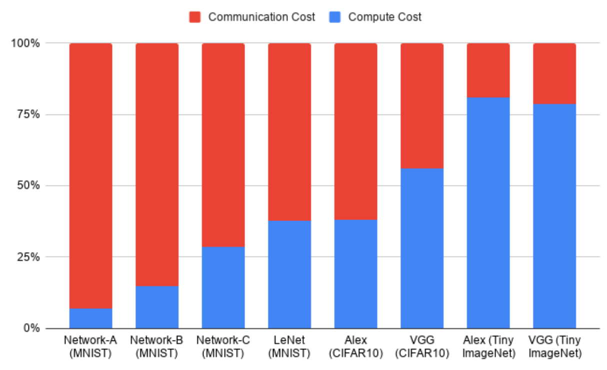


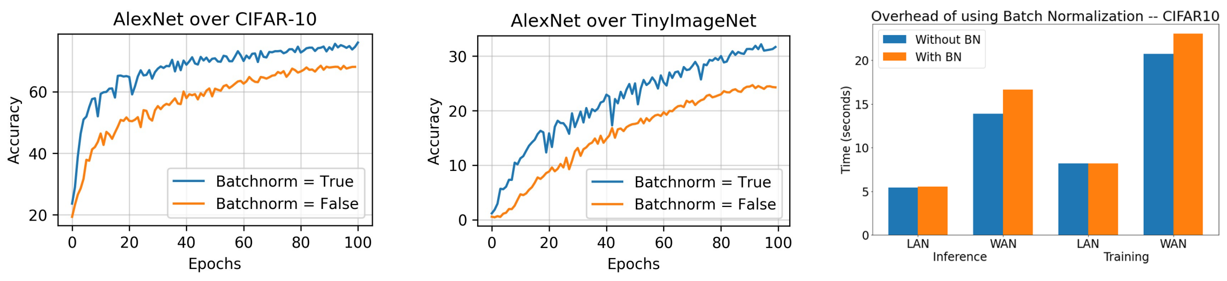
Fig. 1. Compute vs. communication cost for private inference using FALCON over WAN for the malicious adversary. We show that as the network size increases, computation becomes a dominant factor in the overall end-to-end runtime.

$n \times n$ matrices. Hence, for large network architectures, the time required for the matrix-multiplication dominates the overall cost.

This observation is against the conventional wisdom that MPC protocols are communication bound and not computation bound. When running larger networks such as AlexNet and VGG16, and especially for Tiny ImageNet, the computation time starts becoming a significant fraction of the total time. Hence, we claim that FALCON is optimized for communication rounds, specifically when operating over large networks. With our results, we motivate the community to focus on designing faster compute solutions using accelerators such as GPUs, parallelization, efficient matrix multiplications and caching, along with the conventional goals of reducing communication and round complexity.

5.5 Comparison vs. Plaintext Computation

Given the surprising insights from Figure 1, we also compare the execution of privacy-preserving computations with plaintext computations. These results are summarized in Table 7. We use standard PyTorch libraries for the plaintext code, similar hardware as that of privacy-preserving benchmarks for CPU-CPU comparison, and use a single Nvidia P100 GPU for the GPU-CPU comparison. Our findings indicate that private deep learning (over CPU) is within a factor of 40×-



(a) (b) (c)

Fig. 2. In Figs. 2a, 2b, we study the model accuracy with and without batch normalization layers as a function of epochs for AlexNet network. As can be seen, batch normalization not only helps train the network faster but also train better networks. In Fig. 2c, we study the performance overhead of running the network (using FALCON) with and without batch normalization layers.

1200× of plaintext execution of the same network over CPU and within 50×-four orders of magnitude that of plaintext execution over GPU (using PyTorch) when performed over LAN. The overhead further increases by 1.2×-2.4× when comparing against WAN evaluations. This indicates the importance of supporting GPUs and optimizers for private deep learning and showcases the need for further reducing the overhead of MPC protocols. We believe that it is beneficial for the broader research community to have an estimate of the gap between plaintext and privacy-preserving techniques for realistic size networks and datasets.

5.6 Batch Normalization and Accuracy

We study the benefits of batch normalization for privacy-preserving training of neural networks. We compute the accuracy of partially trained models after each epoch with and without the batch normalization layers. As seen in Figs. 2a, 2b, batch normalization layers not only help train the network faster but also train better networks. Fig. 2c demonstrates the overhead of MPC protocols with and without batch normalization layers. Given the high round complexity of batch normalization, the gap is significant only in the WAN setting.

We also study the effect of our approximations and smaller datatype on the accuracy of the computation. We compare the evaluation of the networks with 64-bit `float` datatypes over PyTorch against a 32-bit datatype `uint32_t` using fixed-point arithmetic for FALCON. The final layer outputs differ by small amounts (less than 1%) in comparison with the high precision 64-bit computation. As a consequence, as seen in Table 6, most networks show no/low loss in the overall neural network accuracy when the computation is performed as fixed-point integers over 32-bit datatype. This is because the final prediction is robust to small relative error in individual values at the output. This also makes the final prediction vector inherently noisy and may provide some defense against model inversion attacks.

6 Related Work

Privacy-preserving Training. In a seminal paper on private machine learning, Mohassel *et al.* [4] show protocols for a variety of machine learning algorithms such as linear regression, logistic regression and neural networks. Their approach is based on a 2-party computation model and rely on techniques such as oblivious transfer [46] and garbled circuits [19]. Following that, Mohassel *et al.* [11] proposed a new framework called ABY^3 which generalizes and optimizes switching back and forth between arithmetic, binary, and Yao garbled circuits in a 3-party computation model. Wagh *et al.* [12] proposed SecureNN that considers a similar 3-party model with semi-honest security and eliminate expensive cryptographic operations to demonstrate privacy-preserving training and inference of neural networks. SecureNN also provides *malicious privacy*, a notion formalized by Araki *et al.* [30] but not correctness under malicious corruption. FALCON provides a holistic framework for both training and inference of neural networks while improving computation and communication overhead as compared to prior work.

Privacy-preserving Inference. Privacy-preserving inference has received considerable attention over the last few years. Recall that we have summarized some of these works in Table 1. Private inference typically relies on one or more of the following techniques: secret sharing [11, 12], garbled circuits [7, 9], homomorphic encryption [4, 6, 47] or Goldreich-Micali-Wigderson (GMW) [5, 42], each with its own advantages and disadvantages. CryptoNets [48] was one of the earliest works to demonstrate the use of homomorphic encryption to perform private inference. CryptoDL [49] developed techniques that use approximate, low-degree polynomi-

Run-type			CIFAR-10				Tiny ImageNet			
			Training		Inference		Training		Inference	
			AlexNet	VGG-16	AlexNet	VGG-16	AlexNet	VGG-16	AlexNet	VGG-16
Plaintext	CPU-only	localhost	1.6×10^2	7.3×10^2	7.2×10^1	3.4×10^2	5.0×10^2	3.1×10^3	2.5×10^2	1.3×10^3
	GPU-assisted	localhost	2.8×10^1	6.4×10^1	3.8×10^1	5.8×10^1	3.6×10^1	1.2×10^2	3.8×10^1	5.7×10^1
Private	CPU-only	LAN	6.4×10^3	2.5×10^5	5.6×10^3	1.0×10^5	6.3×10^5	9.5×10^5	2.3×10^5	4.0×10^5
	CPU-only	WAN	2.4×10^4	6.2×10^5	1.7×10^4	1.6×10^5	7.8×10^5	1.2×10^6	3.1×10^5	5.9×10^5
Private	Bandwidth		6.4×10^3	2.5×10^5	5.6×10^3	1.0×10^5	6.3×10^5	9.5×10^5	2.3×10^5	4.0×10^5

Table 7. Comparison of private computation (for semi-honest protocols, cf Section 5.1 for network parameters) with plaintext over the same hardware using PyTorch and a single NVIDIA P100 GPU. Numbers are for a 128 size batch in milliseconds.

als to implement non-linear functions and improve on CryptoNets. DeepSecure [45] uses garbled circuits to develop a privacy-preserving deep learning framework.

Chameleon [5] is another mixed protocol framework that uses the Goldreich-Micali-Wigderson (GMW) protocol [42] for low-depth non-linear functions, garbled circuits for high-depth functions and secret sharing for linear operations to achieve high performance gains. The above three [4, 5, 11] demonstrate private machine learning for other machine learning algorithms such as SVMs, linear and logistic regression as well. Gazelle [8] combines techniques from homomorphic encryption with MPC and optimally balances the use of a specially designed linear algebra kernel with garbled circuits to achieve fast private inference. EzPC [7] is a ABY-based [50] secure computation framework that translates high-level programs into Boolean and arithmetic circuits. Riazi *et al.* propose a framework XONN [9] and showcase compelling performance for inference on large *binarized* neural networks and uses garbled circuits to provide constant round private inference. The work also provides a simple easy-to-use API with a translator from Keras [51] to XONN. EPIC [52] demonstrates the use of transfer learning in the space of privacy-preserving machine learning while Quotient [53] takes the first steps in developing two party secure computation protocols for optimizers and normalizations. CrypTFlow [13] builds on SecureNN and uses trusted hardware to achieve maliciously secure protocols in a 3PC model. Delphi [10] builds on Gazelle to further improve performance and proposes a novel planner that automatically generates neural network architecture configurations that navigate the performance-accuracy trade-offs. Astra [15] is a 3PC protocol with semi-honest security and forms the foundation for a few follow-up works. BLAZE [16] builds on Astra to achieve malicious security and fairness in a 3PC honest majority corruption model and uses an adder circuit approach for non-linear function computation. Trident achieves the same result in a 4PC model with further performance improvements. FLASH [17] also proposes a 4PC model that achieves malicious security with guaranteed output delivery. QuantizedNN [14] proposes an efficient PPML framework using the quantization scheme of Jacob *et al.* [54] and provides protocols in all combinations of semi-honest/malicious security and honest majority vs dishonest majority corruptions.

7 Conclusion

FALCON supports new protocols for private training and inference in a honest-majority 3-party setting. Theoretically, we propose novel protocols that improve the round and communication complexity and provide security against maliciously corrupt adversaries with an honest majority. FALCON thus provides malicious security and provides several orders of magnitude performance improvements over prior work. Experimentally, FALCON is the first secure deep learning framework to examine performance over large-scale networks such as AlexNet and VGG16 and over large-scale datasets such as Tiny ImageNet. We also are the first work to demonstrate efficient protocols for batch-normalization which is a critical component of present day machine learning.

Acknowledgments

We thank Vikash Sehwag for his help with the experiments, the anonymous reviews, and our Shepherd Melek Önen. We also thank the following grant/awards for supporting this work: Facebook Systems for ML award, Qualcomm Innovation Fellowship, Princeton CSML DataX award, Princeton E-ffiliates Partnership award, Army Research Office YIP award, Office of Naval Research YIP Award, and National Science Foundation's CNS-1553437 and CNS-1704105, ISF grant 2774/20, BSF grant 2018393, NSF-BSF grant 2015782, and a grant from the Ministry of Science and Technology, Israel, and the Dept. of Science and Technology, Government of India.

References

[1] E. Bursztein, E. Clarke, M. DeLaune, D. M. Elifff, N. Hsu, L. Olson, J. Shehan, M. Thakur, K. Thomas, and T. Bright, "Rethinking the detection of child sexual abuse imagery on the internet," in *The World Wide Web Conference*. ACM, 2019, pp. 2601–2607.

[2] "Child Abusers Run Rampant as Tech Companies Look the Other Way," https://www.nytimes.com/interactive/2019/11/09/us/internet-child-sex-abuse.html, 2019.

[3] H. Cho, D. J. Wu, and B. Berger, "Secure genome-wide association analysis using multiparty computation," in *Nature biotechnology*, vol. 36, no. 6, 2018, p. 547.

[4] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[5] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[6] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via MiniONN transformations," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[7] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: programmable, efficient, and scalable secure two-party computation for machine learning," in *IEEE European Symposium on Security and Privacy (S&P)*, 2019.

[8] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in *USENIX Security Symposium*, 2018.

[9] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "XONN: XNOR-based oblivious deep neural network inference," in *USENIX Security Symposium*, 2019.

[10] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security Symposium*, 2020.

[11] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[12] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party secure computation for neural network training," in *Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[13] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[14] A. Dalskov, D. Escudero, and M. Keller, "Secure evaluation of quantized neural networks," https://eprint.iacr.org/2019/131, 2019.

[15] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: High throughput 3pc over rings with application to secure prediction," in *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[16] A. Patra and A. Suresh, "Blaze: Blazing fast privacy-preserving machine learning," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[17] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and robust framework for privacy-preserving machine learning," in *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[18] R. Rachuri and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[19] A. C. Yao, "Protocols for secure computations," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[20] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[21] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *Advances in Cryptology—EUROCRYPT*, 2017.

[22] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 35, 2017.

[23] A. Ibarrondo and M. Önen, "Fhe-compatible batch normalization for privacy preserving deep learning," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2018, pp. 389–404.

[24] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *arXiv preprint arXiv:1811.09953*, 2018.

[25] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," https://arxiv.org/abs/1409.1556, 2014.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," 2012.

[27] "MNIST database," http://yann.lecun.com/exdb/mnist/, accessed: 2017-09-24.

[28] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," 2014.

[29] J. Wu, Q. Zhang, and G. Xu, "Tiny ImageNet Challenge," http://cs231n.stanford.edu/reports/2017/pdfs/930.pdf.

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

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

[32] "Announcing securenn in tf-encrypted," https://mc.ai/announcing-securenn-in-tf-encrypted/, 2018.

[33] PySyft, "Implement securenn within pysyft #1990," https://github.com/OpenMined/PySyft/issues/1990, 2019.

[34] "Microsoft photodna cloud service," 2018. [Online]. Available: https://www.microsoft.com/en-us/photodna

[35] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[36] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[37] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," in

USENIX Security Symposium, 2016.

[38] E. Kushilevitz, Y. Lindell, and T. Rabin, "Information-theoretically secure protocols and security under composition," *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2090–2112, 2010.

[39] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[40] O. Catrina and A. Saxena, "Secure computation with fixed-point numbers," in *International Conference on Financial Cryptography and Data Security*, 2010, pp. 35–50.

[41] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele, "Secure computation on floating point numbers," in *Symposium on Network and Distributed System Security (NDSS)*, 2013.

[42] O. Goldreich, S. Micali, and A. 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.

[43] R. Canetti, "Security and composition of multiparty cryptographic protocols," in *Journal of CRYPTOLOGY*, vol. 13, no. 1, 2000, pp. 143–202.

[44] ——, "Universally composable security: A new paradigm for cryptographic protocols," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2001, pp. 136–.

[45] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable provably-secure deep learning," in *Annual Design Automation Conference*, 2018.

[46] C. Peikert, V. Vaikuntanathan, and B. Waters, "A framework for efficient and composable oblivious transfer." in *Advances in Cryptology—CRYPTO*, 2008.

[47] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[48] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*, 2016.

[49] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep Neural Networks over Encrypted Data," in *Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[50] D. Demmler, T. Schneider, and M. Zohner, "ABY – A framework for efficient mixed-protocol secure two-party computation." in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[51] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[52] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: efficient private image classification (or: learning from the masters)," in *Cryptographers' Track at the RSA Conference*. Springer, 2019, pp. 473–492.

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

[54] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[55] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[56] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

A Recent Related Work

We compare the theoretical complexities of protocols in Astra, BLAZE, FLASH, and Trident with protocols in FALCON. Since the approach for computing non-linear operations is fundamentally different, we compare the end-to-end overhead of the ReLU protocol in each of these frameworks. Table 8 shows a comparison of the theoretical complexities. Note that BLAZE and Astra are 3PC protocols and FLASH and Trident are 4PC protocols. In terms of evaluation of neural networks, most of these works evaluate their approach only over DNNs. None of these frameworks evaluate their approaches for *training* of neural networks. Comparison of concrete efficiency of these protocols is documented in Table 2.

B Theoretical Complexity

We theoretically estimate the overheads of our protocols in Table 9. The dominant round complexity for private compare comes from the string multiplication in Step 6. wrap_3 requires one additional round and one additional ring element (two in malicious security) over private compare. Computing derivative of ReLU is a local computation over the wrap_3 function. Computing ReLU requires two additional rounds and one ring element (two for malicious). Maxpool and derivative of require rounds proportional to the area of the filter. Finally, pow, division, and batch-norm require a quadratic number of rounds in ℓ.

C Datasets

We select 3 datasets popularly used for training image classification models — MNIST [27], CIFAR-10 [28], and Tiny ImageNet [29]. We describe these below:

(A) **MNIST [27]:** MNIST is a collection of handwritten digits dataset. It consists of 60,000 images in the training set and 10,000 in the test set. Each image is a 28×28 pixel image of a handwritten digit along with a label between 0 and 9. We evaluate Network-A, B, C, and the LeNet

Protocol	3PC						4PC			
	Astra		BLAZE		FALCON		FLASH		Trident	
	Round	Comm.	Round	Comm.	Round	Comm.	Round	Comm.	Round	Comm.
Multiplication	1	4ℓ	1	3ℓ	1	4ℓ	1	3ℓ	1	3ℓ
ReLU	$3+\log\ell$	45ℓ	4	$(\kappa+7)\ell$	$5+\log\ell$	32ℓ	$\log\ell+10$	46ℓ	4	$8\ell+2$

Table 8. Comparison of theoretical complexities of privacy-preserving protocols for Multiplication and ReLU with Astra, BLAZE, FLASH, Trident, and FALCON. ℓ is the bit-size of the datatype and κ is the security parameter (usually set to 40).

	Protocol	Dependence	Semi-Honest		Malicious	
			Rounds	Comm	Rounds	Comm
Basic Protocols	MatMul	$(x\times y)(y\times z)$	1	kxz	1	$2kxz$
	Private Compare	n	$2+\log_2\ell$	$2kn$	$2+\log_2\ell$	$4kn$
	wrap$_3$	n	$3+\log_2\ell$	$3kn$	$3+\log_2\ell$	$6kn$
Compound Protocols	ReLU and Derivative of ReLU	n	$5+\log_2\ell$	$4kn$	$5+\log_2\ell$	$8kn$
	MaxPool and Derivative of Maxpool	$n,\{w,h\}$	$(wh-1)(7+\log_2\ell)$	$5k+wh$	$(wh-1)(7+\log_2\ell)$	$10k+2wh$
	Pow	n	$5\ell+\ell\cdot\log_2\ell$	$4kn\ell$	$5\ell+\ell\cdot\log_2\ell$	$8kn\ell$
	Division	n	$7+5\ell+\ell\cdot\log_2\ell$	$4kn\ell+7kn$	$7+5\ell+\ell\cdot\log_2\ell$	$8kn\ell+14kn$
	Batch Norm	r,n	$15+5\ell+\ell\cdot\log_2\ell$	$kr+4kr\ell+14krn$	$15+5\ell+\ell\cdot\log_2\ell$	$2kr+8kr\ell+28krn$

Table 9. Theoretical overheads of basic and compound protocols. Communication is in Bytes where ℓ is the logarithm of the ring size and k is its Byte size. We use n to denote the size of the vector in vectorized implementations. Malicious protocols suffer from higher communication complexity compared to semi-honest protocols that results in poor concrete efficiency when implemented.

network on this dataset in both the semi-honest and maliciously secure variants.

(B) **CIFAR-10 [28]:** CIFAR-10 consists of 60,000 images (50,000 training and 10,000 test images) of 10 different classes (such as airplanes, dogs, horses etc.). There are 6,000 images of each class with each image consisting of a colored 32×32 image. We perform private training and inference of AlexNet and VGG16 on this dataset.

(C) **Tiny ImageNet [29]:** Tiny ImageNet dataset consists of 100,000 training samples and 10,000 test samples with 200 different classes [29]. Each sample is cropped to a size of $64\times64\times3$. We perform private training and inference of AlexNet and VGG16 on this dataset.

D Networks

We evaluate FALCON on the following popular deep learning networks. We select these networks based on the varied range of model parameters and different types of layers used in the network architecture. The first three networks are purposely selected to perform performance comparison of FALCON with prior work that evaluated on these models. The number of layers that we report include only convolutional and fully connected layers. We also note that we enable the exact same functionality as prior work with no further approximations. Our networks achieve an accuracy of 97.42% on Network-A, 97.81% on Network-B, 98.64% on Network-C, and 99.15% on LeNet – similar to the accuracy obtained by SecureNN, SecureML, and ABY^3 [4, 11, 12].

(A) **Network-A:** This is a 3 layer fully-connected network with ReLU activation after each layer as was evaluated in SecureML [4]. This is the smallest network with around 118K parameters.

(B) **Network-B:** This network is a 3 layer network with a single convolution layer followed by 2 fully-connected layers and ReLU activations. This architecture is chosen from Chameleon [5] with approximately 100K parameters.

(C) **Network-C:** This is a 4 layer network with 2 convolutional and 2 fully-connected layers selected from prior work MiniONN [6]. This network uses Max Pooling in addition to ReLU layer and has around 10,500 parameters in total.

(D) **LeNet:** This network, first proposed by LeCun et al. [55] was used in automated detection of zip codes and digit recognition [56]. The network contains 2 convolutional layers and 2 fully connected layers with 431K parameters.

(E) **AlexNet:** AlexNet is the famous winner of the 2012 ImageNet ILSVRC-2012 competition [26]. It has 5 convolutional layers and 3 fully connected layers and uses batch norm layer for stability, efficient training and has about 60 million parameters. FALCON is the first private deep learning framework that evaluates AlexNet because of the support for batch norm layer in our system.

(F) **VGG16:** The last network which we implement is called VGG16, the runner-up of the ILSVRC-2014 competition [25]. VGG16 has 16 layers and has about 138 million parameters.

E Functionality Descriptions

$\mathcal{F}_{\mathsf{PC}}$

Input: The functionality receives inputs $\{[\![x_i]\!]^p\}_{i=1}^{\ell}, r$

Output: Compute the following

1. Reconstruct bits x_i and $x = \sum x_i \cdot 2^i$
2. Compute $b = (x \geq r)$
3. Generate random shares of b and send back to the parties

Fig. 3. Ideal functionality for Π_{PC}

$\mathcal{F}_{\mathsf{WA}}$

Input: The functionality receives inputs $[\![a]\!]^L$.

Output: Compute the following

1. Compute $b = \mathsf{wrap}_3(a_1, a_2, a_3, L)$
2. Generate random shares of b and send back to the parties

Fig. 4. Ideal functionality for Π_{WA}

$\mathcal{F}_{\mathsf{ReLU}}$

Input: The functionality receives inputs $[\![a]\!]^L$.

Output: Compute the following

1. Compute $b = \mathsf{ReLU}(a_1 + a_2 + a_3 \pmod L)$
2. Generate random shares of b and send back to the parties

Fig. 5. Ideal functionality for Π_{ReLU}

Π_{Prep}

Usage: This is used to generate pre-processing material required for the online protocol.

Setup: This step will have to be done only once.

1. Each party P_i chooses a random seed k_i
2. Send this random seed to party P_{i+1}

Common randomness: Let F be any seeded PNRG. Then 3-out-of-3 and 2-out-of-3 common randomness described in Section 3.2 can be generated as follows:

1. $\alpha_i = F_{k_i}(\mathsf{cnt}) - F_{k_{i-1}}(\mathsf{cnt})$ and cnt++
2. $(\alpha_i, \alpha_{i-1}) = (F_{k_i}(\mathsf{cnt}), F_{k_{i-1}}(\mathsf{cnt}))$ and cnt++

Truncation Pair: Generate truncation pair $[\![r]\!], [\![r']\!] = [\![r/2^d]\!]$.

1. Run protocol Π_{trunc2} from [11] (Figure 3)

Correlated randomness for Private Compare: Generate correlated randomness required for Π_{PC}

1. Sample random bit $[\![b]\!]^2$
2. Use bit injection from [11] $[\![b]\!]^2 \to [\![b]\!]^p$
3. Sample random values $m_1, \dots m_k \in \mathbb{Z}_p$.
4. Compute and open $m_1^{p-1}, \dots, m_k^{p-1}$.
5. Remove openings that equal 0 and queue openings that equal 1. Note that this computation takes $\lceil \log_2 p \rceil$ rounds and can be amortized for efficiency (by setting a large value of k).

Correlated randomness for Wrap$_3$: Generate correlated randomness required for Π_{WA}

1. Sample random bits $[\![r_i]\!]^2$ for $i \in [\ell]$
2. Perform bit composition from [11] to get $[\![r_i]\!]^L$
3. Use bit injection from [11] $[\![r_i]\!]^2 \to [\![r_i]\!]^p$
4. Use the optimized full adder FA to compute the final carry bit. Note that this bit is precisely $\mathsf{wrap}_3(\cdot)$

Correlated randomness for ReLU: Generate correlated randomness required for Π_{ReLU}

1. Sample random bit $[\![b]\!]^2$
2. Use bit injection from [11] $[\![b]\!]^2 \to [\![b]\!]^L$

Correlated randomness for Maxpool and Division: No additional correlated randomness necessary other than that used in their subroutines.

Fig. 6. Protocols for generating various pre-processing material

$\mathcal{F}_{\text{Maxpool}}$

Input: The functionality receives inputs $[\![a_1]\!]^L, \ldots [\![a_n]\!]^L$.

Output: Compute the following

1. Reconstruct $a_1, \ldots a_n$ and compute $k = \text{argmax}\{a_1, \ldots a_n\}$.
2. Set $\boldsymbol{e_k} = \{e_1, e_2, \ldots e_n\}$ with $e_i = 0\ \forall i \neq k$ and $e_k = 1$.
3. Generate random shares of a_k and $\boldsymbol{e_k}$ and send back to the parties.

Fig. 7. Ideal functionality for Π_{Maxpool}

$\mathcal{F}_{\text{Pow}}$

Input: The functionality receives inputs $[\![b]\!]^L$ and an index $k \in \{0, 1, \ldots \ell - 1\}$.

Output: Compute each bit of α sequentially as follows:

1. Reconstruct b.
2. Compute α such that $2^{\alpha-1} < b \leq 2^{\alpha}$
3. If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
4. If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.

Fig. 8. Functionality for Π_{Pow}

$\mathcal{F}_{\text{Div}}$

Input: The functionality receives inputs $[\![a]\!]^L, [\![b]\!]^L$ and an index $k \in \{0, 1, \ldots \ell - 1\}$.

Output: Compute the following

1. Reconstruct a, b.
2. Compute α such that $2^{\alpha} \leq b < 2^{\alpha+1}$
3. If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
4. If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.
5. Generate random shares of $a \cdot \text{AppDiv}(b)$ and send back to the parties

Fig. 9. Ideal functionality for Π_{Div}

$\mathcal{F}_{\text{BN}}$

Input: The functionality receives inputs $[\![a_1]\!]^L, \ldots [\![a_n]\!]^L$ and $[\![\gamma]\!]^L, [\![\beta]\!]^L$ and an index $k \in \{0, 1, \ldots \ell - 1\}$.

Output: Compute the following

1. Reconstruct $a_1, \ldots a_n$ and compute μ and σ^2 as given in Step 1,2 of Algorithm 7
2. Set $b = \sigma^2 + \epsilon$ and compute α such that $2^{\alpha} \leq b < 2^{\alpha+1}$
3. If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
4. If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.
5. Complete steps 4-8 of Algorithm 7 and return random shares of the output.

Fig. 10. Ideal functionality for Π_{BN}