Adam in Private: Secure and Fast Training of Deep Neural Networks with Adaptive Moment Estimation

Nuttapong Attrapadung*, Koki Hamada†, Dai Ikarashi†, Ryo Kikuchi†, Takahiro Matsuda*,
Ibuki Mishina†, Hiraku Morita‡ and Jacob C. N. Schuldt*
*National Institute of Advanced Industrial Science and Technology
Email: {n.attrapadung, t-matsuda, jacob.schuldt}@aist.go.jp
†NTT
Email: kikuchi_ryo@fw.ipsj.or.jp, {koki.hamada.rb, dai.ikarashi.rd, ibuki.mishina.br}@hco.ntt.co.jp
‡University of St. Gallen
Email: hiraku.morita@unisg.ch

***Abstract*—Machine Learning (ML) algorithms, especially deep neural networks (DNN), have proven themselves to be extremely useful tools for data analysis, and are increasingly being deployed in systems operating on sensitive data, such as recommendation systems, banking fraud detection, and healthcare systems. This underscores the need for privacy-preserving ML (PPML) systems, and has inspired a line of research into how such systems can be constructed efficiently. We contribute to this line of research by proposing a framework that allows efficient and secure evaluation of full-fledged state-of-the-art ML algorithms via secure multi-party computation (MPC). This is in contrast to most prior works on PPML, which require advanced ML algorithms to be substituted with approximated variants that are "MPC-friendly", before MPC techniques are applied to obtain a PPML algorithm. A drawback of the latter approach is that it requires careful fine-tuning of the combined ML and MPC algorithms, and might lead to less efficient algorithms or inferior quality ML (such as lower prediction accuracy). This is an issue for secure training of DNNs in particular, as this involves several arithmetic algorithms that are thought to be "MPC-unfriendly", namely, integer division, exponentiation, inversion, and square root extraction.**

In this work, we propose secure and efficient protocols for the above seemingly MPC-unfriendly computations (but which are essential to DNN). Our protocols are three-party protocols in the honest-majority setting, and we propose both passively secure and actively secure with abort variants. A notable feature of our protocols is that they simultaneously provide high accuracy and efficiency. This framework enables us to efficiently and securely compute modern ML algorithms such as Adam (Adaptive moment estimation) and the softmax function "as is", without resorting to approximations. As a result, we obtain secure DNN training that outperforms state-of-the-art three-party systems; our *full* training is up to 6.7 times faster than just the *online* phase of the recently proposed FALCON (Wagh et al. at PETS'21) on the standard benchmark network for secure training of DNNs. To further demonstrate the scalability of our protocols, we perform measurements on real-world DNNs, AlexNet and VGG16, which are complex networks containing millions of parameters. The performance of our framework for these networks is up to a factor of about $12 \sim 14$ faster for AlexNet and $46 \sim 48$ faster for VGG16 to achieve an accuracy of 70% and 75%, respectively, when compared to FALCON.

I. Introduction

Secure multi-party computation (MPC) [58], [21], [5] enables function evaluation, while keeping the input data secret. An emerging application area of secure computation is privacy-preserving machine learning (ML), such as (secure) deep neural networks. Combining secure computation and deep neural networks, it is possible to gather, store, train, and derive predictions based on data, which is kept confidential. This provides data security and encourages data holders to share their confidential data for machine learning. As a consequence, it becomes possible to use a large amount of data for model training and obtain accurate predictions.

We first briefly review a typical *training* (or learning) process of a deep neural network in the clear (*i.e.,* without secure computation). A deep neural network (DNN) consists of several layers, and certain functions are sequentially computed on the training data layer-by-layer. The so-called *softmax* function is one of the more common functions computed in the last layer. Then, a tentative output from the last layer is computed, and a convergence test is applied to this. Based on the result of the test, the parameters are updated by an optimization method, and the above processes will be repeated. A traditional optimization method is *stochastic gradient descent (SGD)*. As SGD tends to incur many repetitions (and hence slow convergence), more efficient approaches have been proposed; adaptive gradient methods such as *adaptive moment estimation (Adam)* [29] are popular optimization methods which improve upon SGD and are adopted in many real-world tool-kits, *e.g.,* [53].

A key challenge towards privacy-preserving ML, especially for DNN, is how to securely compute functions that are *not* "MPC-friendly". MPC-friendly functions refer to functions that are easy to securely compute in MPC, and for which very efficient protocols exist. However, unfortunately, functions required in DNN are often *MPC-unfriendly*, especially those used in more modern approaches to training. In particular, Adam [29] (and also the softmax function) consist of several MPC-unfriendly functions, namely, integer division, exponentiation, inversion, and square root computations.

To cope with this challenge, up to now, there have been two lines of research. First, many works (to name just a few, [19], [37], [9], [7], [50], [51], [10], [30], [45], [8], [31]) have focused mainly on secure protocols for the *prediction* (or inference) process only, which is much more lightweight compared to the *training*, as gradient optimization methods

are not required for prediction. Second, and more recently, there have been a few works in the literature that can handle secure training. These are done mostly by replacing originally MPC-unfriendly functions with different ones that are *MPC-friendly* and approximate the original function on the domain of interest. These approximation approaches either can be done only for elementary optimization methods such as SGD, as in [43], [56], [42], [11] or require specific "fine-tuning" of the interaction between ML and MPC, as in [1], such that the replaced functions will not degrade the quality of ML architectures significantly (such as lowering prediction accuracy). In practice, however, this replacement is not easy. For example, Keller and Sun [25] reported that ASM, which is widely used as a replacement for the softmax function, reduces accuracy in training, sometimes significantly.

Due to the rapid advancements in ML, we believe that a more robust approach to privacy-preserving ML is to achieve efficient protocols for a set of functions that are often used in ML but might typically be thought of as MPC-unfriendly. In this way, the requirement for fine-tuning between ML and MPC would be only minimal, if any at all, and one would be able to plug-and-play new ML advancements into an existing MPC framework to obtain new privacy-preserving ML protocols, without having to worry about the degradation on the ML side.

A. Our Contributions

We present a framework that allows seamless implementation of secure training for DNNs using modern ML algorithms. Specifically, our contribution is twofold as follows.

New Elementary Three-party Protocols. We propose new secure and efficient protocols for a set of elementary functions that are useful for DNN but are normally deemed to be MPC-unfriendly. These include secure division, exponentiation, inversion, and square root extraction. Our protocols are three-party protocols in the honest-majority setting, and we propose both passively secure and actively secure with abort variants. A notable feature of our protocols is that they simultaneously provide high accuracy and efficiency. A key component to this is our new division protocol, which enables secure fixed-point arithmetic. To the best of our knowledge, all previous direct fixed-point arithmetic protocols introduce errors with some probability which must be mitigated, typically resulting in an increased overhead or reduced accuracy. In contrast, no such mitigation step is required for our protocols. Combined with a range of optimizations suitable for each of the functionalities we consider, we obtain a set of protocols that are both very efficient and ensure high accuracy. In fact, our efficient implementations of our protocols provide 23-bit accuracy fixed-point arithmetic, which is comparable to single-precision real number operations *in the clear*. We discuss our construction techniques further in the following section.

New Applications to ML. We apply our new elementary MPC protocols to "seamlessly" instantiate secure computations for softmax and Adam. That is, due to our elementary MPC protocols, we can securely and efficiently compute softmax and Adam "as is", in particular, without approximation using (MPC-friendly) functions. Consequently, due to the fast convergence of Adam, we obtain fast and secure training (and prediction) protocols for DNN. Using the DNN architecture and MNIST dataset typically used as a benchmark, our protocol achieved 95.64% accuracy within 117 seconds, improving upon the state-of-the-art such as ABY3 [42] (94% accuracy within 2700 seconds reported in [42]) and FALCON [57] (780 seconds for the online phase only)[1]. Moreover, our training converges much faster, namely, in one epoch, as opposed to 15 epochs for ABY3 and FALCON. Furthermore, our protocol achieves the same accuracy as training over *plaintext* data, using MLPClassifier in the scikit-learn tool-kit [46] while being less than six times slower. We further perform measurement on real-world DNNs from the ML literature, AlexNet [33] and VGG16 [54], which contain millions of parameters. Comparing the *total training time* (i.e. time to reach a certain accuracy), the total running time of our framework outperforms the online phase of FALCON with a factor of about $12 \sim 14$ for AlexNet and $46 \sim 48$ for VGG16 in the LAN setting. A detailed performance evaluation and comparison considering different security and network settings, different datasets, and large DNNs, is given in Section VI.

B. Our Techniques

New Techniques for Secure Truncation. We first briefly describe the idea behind a common building block for all our protocols: division (which also implies truncation). Let p be the size of the underlying ring/field, x be the secret and d is the divisor (so the desired output is $\frac{x}{d}$). Known efficient truncation protocols, *e.g.,* [42], [43], reconstruct a masked secret $x + r$ for a random r, divide this by d in the clear, and subtract $\frac{r}{d}$. However, in this approach, a large error, $-\frac{p}{d}$, sneaks into the output when $x + r > p$ because the reconstructed value becomes $x + r - p$. To avoid this, the message space has to be much smaller than p, which leads to reduced accuracy for a given value of p. Instead, we employ a different approach. Let x_1 and x_2 be additive shares of x such that $x_1 + x_2 = x + qp$ for $q \in \{0, 1\}$. Our approach is to securely compute q and eliminate qp (without exposing q to any parties), which makes the (local) division of sub-shares be the desired output. In this way, we can embed a large value into a single share, which, in turn, enables accurate computation of functions such as exponentiation.

New Techniques for Elementary Protocols. For securely computing exponentiation, inversion, division with private divisor, square root, and inversion of square root, we utilize Taylor or Newton series expansions. A key challenge here is to ensure fast convergence that, in general, is only guaranteed for a narrow range of input values. We resolve this by constructing protocols that use a combination of private input pre-processing and partial evaluation of the pre-processed input. We devise *private scaling* techniques, which allow inputs to be scaled to fit an optimal input range, and furthermore allow the protocol to make the most out of the available bit range in the internal computations. We also utilize what we call *hybrid table-lookup/series-expansion* techniques, which separate inputs into two parts and apply table-lookup and series-expansion to the respective parts. The details of how these techniques are used in our protocols differ depending

[1]The measurements for our protocol and FALCON were done in the environment described in Section VI, which is roughly comparable to the one in [42].

on the functionality of the protocols. We provide detailed descriptions in Section IV.

C. Related Work

Various ML algorithms have been considered in connection with privacy preserving ML, include decision trees, linear regression, logistic regression, support-vector-machine classifications, and deep neural networks (DNN). Among these, deep neural networks are the most flexible and have yielded the most impressive results in the ML literature. However, at the same time, secure protocols for DNN are the most difficult to obtain, especially for the training process. We show a table for comprehensive comparison among PPML systems supporting DNN in Table I.

Secure DNN Training. Our work focuses on *secure training for deep neural networks* (secure inference can be obtained as a special case). There have been several works on secure DNN training such as SecureML [43], SecureNN [56], ABY3 [42], Quotient [1], FHE-based SGD [44], Glyph [39], Trident [11], and FALCON [57]. All of these achieve efficiency by simplifying the underlying DNN training algorithms (e.g. replacing functionalities with less-accurate easier-to-compute alternatives), and optimizing the computation of these. As a consequence of this approach, they are restricted to simple SGD optimization, with the exception of Quotient which implements an approximation to AMSGrad. We emphasize that we take a fundamentally different approach by constructing protocols that allow unmodified advanced training to be done efficiently. In the following, we highlight properties of the above related works.

Setting/Security. SecureML, Quotient, FHE-based SGD, and Glyph are two-party protocols, SecureNN, ABY3, and FALCON are three-party protocols, while Trident is a four-party protocol in a somewhat unusual asymmetric offline-online setting. SecureML, Quotient, FHE-based SGD, Glyph, and SecureNN considered *semi-honest (passive)* security tolerating one corrupted party, while SecureNN can be extended to achieve so-called *privacy against malicious adversaries* (formalized by [4]). ABY3 improved security upon these by considering *malicious (active) security with abort* tolerating one corrupted party. Trident improved security in term of *fairness* (again, tolerating one corrupted party); this comes with the cost of reducing the tolerated corruption fraction from 33% to 25%. It should be noted that, unlike the other schemes, FALCON sacrifice perfect security to compute batch-normalization more efficiently (see Section V-B).

Efficiency. For secure training over a basic 3-layer DNN on the MNIST dataset, ABY3 outperforms both SecureML/SecureNN and was state-of-the-art before Trident and FALCON. FHE-based SGD and Glyph use fully homomorphic encryption, which makes non-interactive training possible. Glyph is the most efficient of the two, but is still far less efficient than ABY3 in terms of execution time. Trident improves the *online* phase of ABY3 but with the cost of adding a fourth party who only participates in offline phase. Most recently, FALCON also improves upon the *online* phase of ABY3. As highlighted above, our framework improves upon FALCON.

Additional Related Works. Note that when considering only secure *inference* (but not secure training) for DNN, BLAZE [45] achieved stronger security (than that of ABY3) of *fairness*.

For the less flexible ML algorithms, namely, linear/logistic regression (which we do not focus on in this work), there have been recent progresses in secure training. Up to 2018, ABY3 was the state-of-the-art in terms of performance and security, achieving the same security as their DNN counterpart. Recently, for linear/logistic regression, BLAZE [45] improved ABY3 by 50-2600 times in performance and also achieved *active security with fairness*. More recently, SWIFT [31] achieved the strongest notion, namely, *active security with guaranteed output delivery*.

II. Preliminaries and Settings

Notations for Division. For $a, b \in \mathbb{Z}$, we denote by $\frac{a}{b} \in \mathbb{R}$ real-valued division, and by $a/b \in \mathbb{Z}$ integer division that discards the remainder. In other words, $a/b := \lfloor \frac{a}{b} \rfloor$.

A. Data Representation

The data representation is an important aspect of efficient and accurate computation. The algorithms considered in this paper make use of the following data types:

- Binary values $\mathbb{Z}_2$.
- ℓ-bit unsigned and signed integers, $\mathbb{Z}^+_{\langle \ell \rangle} = \{a \in \mathbb{Z} \mid 0 \leq a \leq 2^\ell - 1\}$ and $\mathbb{Z}_{\langle \ell \rangle} = \{a \in \mathbb{Z} \mid -2^{\ell-1} \leq a \leq 2^{\ell-1} - 1\}$, respectively, where the range of values for signed integers reflect that a single bit is used to indicate the sign.
- ℓ-bit fixed-point unsigned and signed rational numbers $\mathbb{Q}^+_{\langle \ell,u \rangle} = \{b \in \mathbb{Q} \mid b = \frac{a}{2^u}, a \in \mathbb{Z}^+_{\langle \ell \rangle}\}$ and $\mathbb{Q}_{\langle \ell,u \rangle} = \{b \in \mathbb{Q} \mid b = \frac{a}{2^u}, a \in \mathbb{Z}_{\langle \ell \rangle}\}$, respectively.

We represent these data types as follows. Binary values are represented as is, i.e. as elements of the field $\mathbb{F} = \mathbb{Z}_2$. Signed and unsigned integers are represented as elements of the field $\mathbb{F} = \mathbb{Z}_p$ for a Mersenne prime $p > 2^k$. This implies that signed integers are represented using ones' compliment (i.e. a negative value $-a \in \mathbb{Z}_{\langle \ell \rangle}$ is represented as $p - a$, and the most significant bit, which indicates the sign, will be 1). We will likewise represent fixed-point values as elements of $\mathbb{F}_p$, and in order to do so, these are scaled to become integers. Specifically, we will use a set of (unsigned) ℓ-bit integers $0 \leq a \leq 2^\ell - 1$, which we denote $\widehat{\mathbb{Q}}^+_{\langle \ell,\alpha \rangle}$, to represent the values $\{\frac{a}{2^\alpha} | 0 \leq a \leq 2^\ell - 1\}$, and will refer to α as the *offset* for these. For a fixed-point value a, we will use the notation $a_{\langle \alpha \rangle}$ to denote the integer representation with offset α i.e. $a_{\langle \alpha \rangle} = a \cdot 2^\alpha$. The integers in $\widehat{\mathbb{Q}}^+_{\langle \ell,\ell \rangle}$ are represented as elements of $\mathbb{F}_p$, and we denote the signed extension by $\widehat{\mathbb{Q}}_{\langle \ell,\ell \rangle}$.

Note that the representation of fixed-point values requires the scaling factor to be taken into account for multiplication (and division). Specifically, for values $a_{\langle \alpha \rangle}$ and $b_{\langle \alpha \rangle}$, the correct representation of the product of a and b is $a_{\langle \alpha \rangle} \cdot b_{\langle \alpha \rangle} / 2^\alpha = (a \cdot b)_{\langle \alpha \rangle}$. For simplicity, we use $\times_\alpha$ to denote this operation, i.e. (ordinary) multiplication followed by division by 2^α.

Finally note that since $p = 2^\lambda - 1$ is a Mersenne prime, modular arithmetic in $\mathbb{F}_p$ can be done swiftly via bit-shifting and addition (e.g. see [6]). Specifically, if $a = a_0 2^\lambda + a_1$, then

TABLE I: Comparison among various privacy-preserving ML systems.

	System	Prediction	Training	Basic	Batch-Norm	Advance (ex. Adam)	Semi-honest	Malicious	HE	GC	SS	LAN	WAN	Small (ex. MNIST)	Large (ex. CIFAR-10)	Simple (ex. 3DNN)	Complex (ex. VGG-16)
		Secure Capability		Supported ML Algorithms			Threat Model		Based Techniques			LAN/ WAN		Evaluation Dataset		Network Architectures	
		Theoretical metric										Evaluation metric					
2PC	MiniONN [37]	●	○	●	○	○	●	○	●	●	●	●	○	●	●	●	○
	Chameleon [50]	●	○	●	○	○	●	○	○	●	●	●	●	●	●	●	◐
	EzPC [9]	●	○	●	○	○	●	○	○	●	●	●	●	●	●	●	○
	Gazelle [24]	●	○	●	○	○	●	○	●	●	●	●	○	●	●	●	○
	SecureML [43]	●	●	●	○	○	●	○	●	●	●	●	◐	●	○	●	○
	XONN [49]	●	○	●	◐	○	●	●	○	●	●	●	○	●	●	●	◐
	Quotient [1]	●	●	●	●	●	●	○	○	●	●	●	●	●	○	●	◐
	Delphi [41]	●	○	●	○	○	●	○	●	●	●	●	○	○	●	●	◐
	FHE-based SGD [44]	●	●	●	●	○	●	○	●	○	○	●	○	●	○	●	○
	Glyph [39]	●	●	●	●	○	●	○	●	○	○	●	○	●	●	●	○
3PC	ABY3 [42]	●	●	●	○	○	●	●	○	●	●	●	◐	●	○	●	○
	SecureNN [56]	●	●	●	◐	○	●	◐	○	○	●	●	●	●	○	●	○
	CryptFlow [34]	●	○	●	○	○	●	●	○	○	●	●	○	●	●	●	◐
	QuantizedNN [16]	●	○	●	◐	○	●	●	●	○	●	●	●	○	●	○	◐
	ASTRA [10]	●	○	●	○	○	●	●	○	●	●	●	●	●	○	●	○
	BLAZE [45]	●	◐	●	○	○	●	●	○	●	●	●	●	●	○	●	○
	Falcon [57]	●	●	●	●	○	●	●	○	○	●	●	●	●	●	●	●
	This work	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●
4PC	FLASH [8]	●	○	●	○	○	●	●	○	○	●	●	●	●	◐	●	○
	Trident [47]	●	●	●	○	○	●	●	○	●	●	●	●	●	○	●	○

"Basic" for Supported ML Algorithms refers to more basic ones such as linear operations, convolution, ReLU, Maxpool, and/or SGD optimizer. "Advance" refers to advance optimizers, namely, ADAM (considered in this work) and AMSGrad (in Quotient). HE, GC, SS refer to homomorphic encryption, garbled circuit, and secret sharing, respectively. "Small" for Evaluation Dataset refers to MNIST, except for BLAZE, which uses Parkinson disease dataset and for Quotient, which uses also MotionSense, Thyroid, and more, besides MNIST (their dimensions are similar to MNIST). "Large" refers to larger datasets such as the well-known CIFAR-10 in particular (in all the systems that tick except QuantizedNN), or TinyImageNet (in CryptFlow and QuantizedNN, and partially in Falcon). "Simple" for Network Architectures refers to simple neural networks such as the basic 3-layer DNN (3DNN) from SecureML in particular, or other slightly different small networks from [37], [56]. "Complex" refers to more complex networks such as the well-known AlexNet and VGG-16 in particular (both are considered in Falcon and this work, while XONN uses VGG-16 among other networks). ● indicates that such a system support a feature, ○ indicates that such a system does not so support so, ◐ refers to fair comparison being difficult due to various reasons. Secure training in BLAZE only considers the case for less advance ML algorithms *e.g.,* linear/logistic regression, but notably not neural networks. (Their secure prediction, on the other hand, includes neural networks). XONN, QuantizedNN support simplified or different versions of batch normalization, while SecureNN supports divisions but not batch norm. SecureML only estimate their WAN evaluation, while ABY3 does not present WAN results for neural networks. SecureNN achieves malicious privacy (but not including correctness), as defined in [4]. FLASH uses a smaller data set than CIFAR-10. Networks with moderate sizes are experimented: *e.g.,* ResNet-20 (in Quotient), ResNet-32 (in Delphi), ResNet-50, DenseNet-121 (in CryptFlow), MobileNet (in QuantizedNN). Chameleon uses a slightly weaker version of AlexNet.

$a \mod p = a_0 2^\lambda + a_1 \mod p = a_0 + a_1 \mod p$ holds since $2^\lambda - 1 = 0 \mod p$. This will allow efficient operations on shared integers or fixed-point values represented as above.

B. Multi-Party Computation Setting

We consider secret-sharing (SS)-based three-party computation secure against a single static corruption: There are three parties P_1, P_2, P_3, a secret is shared among these parties via SS, any two parties can reconstruct the secret from their shares, and an adversary corrupts up to a single party at the beginning of the protocol. For notational convenience, we treat the party index $i \in \mathbb{Z}$ as to refer to the i'-th party where $i' \equiv i \pmod 3$ and $i' \in \{1, 2, 3\}$. For example, $P_0 = P_3$ and $P_4 = P_1$.

We consider the client/server model. This model is used to outsource secure computation, where any number of clients send shares of their inputs to the servers. Hence, both the input and output of the servers are in a secret-shared form, and our protocols are thus share-input and share-output protocols. More precisely, during secure training, the three parties have shares of training data as input, and then interact with each other to obtain *shares* of a trained model. This setting is composable, i.e., there is a degree of freedom in how the input and output come in and how they are used. For example, a client other than the parties may provide input, or the output of another secure protocol may be used as an input. The resulting model can be made public, or the prediction can be made while maintaining the model secret.

Regarding the adversarial behavior, we consider both passive (semi-honest) and active (malicious) adversaries with abort. In passive security, corrupted parties follow the protocol but might try to obtain private information from the transcripts of messages that they receive. Formally, we say that a protocol is passively secure if there is a simulator that simulates the view of the corrupted parties from the inputs and outputs of the protocol [20]. In active security with abort, corrupted parties are allowed to behave arbitrarily in attempt to break the protocol. The probability an active adversary successfully cheats is parametarized by the statistical security parameter κ, meaning that probability is bounded by $2^{-\kappa}$.We prove the security of our protocols in a hybrid model, where parties run the real protocols, but also have access to a trusted party computing specified subfunctionalities for them. For a subfunctionality denoted g, we say that the protocol runs in the g-hybrid model.

C. Secret Sharing Schemes and Their Protocols

In this paper, we use three *replicated secret sharing schemes* [23], [15]. We consider the 2-out-of-3 threshold access structure for the first two schemes. For the third scheme, the minimal access structure is simply $\{\{1,2\}\}$, meaning only P_1 and P_2 can together reconstruct the secret. We denote them as:

$\llbracket\cdot\rrbracket$-sharing : the 2-out-of-3 replicated sharing in $\mathbb{Z}_p$,
$[\cdot]$-sharing : the 2-out-of-3 replicated sharing in $\mathbb{Z}_2$,
$\langle\langle\cdot\rangle\rangle$-sharing : the simple additive sharing in $\mathbb{Z}_p$.

$\llbracket\cdot\rrbracket$-sharing. This scheme is specified by:

- Share: To share $a \in \mathbb{Z}_p$, pick random $a_1, a_2, a_3 \in \mathbb{Z}_p$ such that $a = a_1 + a_2 + a_3$, then set $\llbracket a\rrbracket_i = (a_i, a_{i+1})$ as the P_i's share for $i = 1, 2, 3$. Denote $\llbracket a\rrbracket = (\llbracket a\rrbracket_1, \llbracket a\rrbracket_2, \llbracket a\rrbracket_3)$.
- Reconstruct: Given a pair of shares of a, this protocol with passive adversary guarantees that all the parties eventually obtain a. With an active adversary, this functionality proceeds the same unless $\llbracket a\rrbracket$ is not consistent, where all the honest parties will abort at the end of the execution.
- Local operations: Given shares $\llbracket a\rrbracket$ and $\llbracket b\rrbracket$ and a scalar $\alpha \in \mathbb{Z}_p$, the parties can generate shares of $\llbracket a+b\rrbracket$, $\llbracket \alpha a\rrbracket$, and $\llbracket \alpha + a\rrbracket$ using only local operations. The notations $\llbracket a\rrbracket + \llbracket b\rrbracket$, $\alpha\llbracket a\rrbracket$, and $\alpha + \llbracket a\rrbracket$ denote these local operations, respectively.
- Multiplications: Given shares $\llbracket a\rrbracket$ and $\llbracket b\rrbracket$, the parties can generate $\llbracket ab\rrbracket$ by a multiplication protocol [17], [18], [14]. We denote this functionality as $\mathcal{F}_{\text{mult}}$.

$[\cdot]$-sharing. This is exactly as $\llbracket\cdot\rrbracket$-sharing but with $p = 2$.

Combining local operations with multiplication protocols, the parties can compute any arithmetic/boolean circuit over shared data, *e.g.,* the parties obtain $[a \wedge b]$ by multiplying $[a]$ and $[b]$ via $\mathcal{F}_{\text{mult}}$.

$\langle\langle\cdot\rangle\rangle$-sharing. To share $a \in \mathbb{Z}_p$, pick random $\langle\langle a\rangle\rangle_1, \langle\langle a\rangle\rangle_2 \in \mathbb{Z}_p$ such that $a = \langle\langle a\rangle\rangle_1 + \langle\langle a\rangle\rangle_2$. Set $\langle\langle a\rangle\rangle_3$ as the empty string. $\langle\langle a\rangle\rangle_i$ is the P_i's share. Denote $\langle\langle a\rangle\rangle = (\langle\langle a\rangle\rangle_1, \langle\langle a\rangle\rangle_2, \langle\langle a\rangle\rangle_3)$.

Share Conversions. Our protocols will utilize conversions among sharing types. Due to limited space, we defer the details to Appendix A, and provide a summary in Table II below. Here, for $a \in \mathbb{Z}_p$ we let $(a_\ell, \ldots, a_1)$ be the bit representation of a; that is, $a = \sum_{i=1}^{\ell} 2^{i-1} a_i$. Round is of a passively secure protocol.

TABLE II: Share Conversions

Conversion	Functionality name	Protocol	Round
$\llbracket a\rrbracket \to \langle\langle a\rangle\rangle$	ConvertToAdd	local operations	0
$\langle\langle a\rangle\rangle \to \llbracket a\rrbracket$	ConvertToRep	one $\llbracket\cdot\rrbracket$-sharing	1
$\llbracket a\rrbracket \to ([a_\ell], \ldots, [a_1])$	$\mathcal{F}_{\text{BDC}}$ – Bit decomposition	[28]	$\ell + 1$
$([a_\ell], \ldots, [a_1]) \to \llbracket a\rrbracket$	$\mathcal{F}_{\text{BC}}$ – Bit composition	[2] (modified)	$\ell + 1$
$[a] \to \llbracket a\rrbracket$	$\mathcal{F}_{\text{mod}}$ – Modulus conversion	[28]	1

Conditional Assignment. We define a functionality of conditional assignment $\llbracket z\rrbracket \leftarrow \mathcal{F}_{\text{CondAssign}}(a, b, [c])$ via setting $z := a$ if $c = 0$ and $z := b$ if $c = 1$. A protocol for this simply converts $\llbracket c\rrbracket := \mathcal{F}_{\text{mod}}([c])$, and computes $\llbracket z\rrbracket := a \cdot (1 - \llbracket c\rrbracket) + \llbracket c\rrbracket \cdot b$.

D. Quotient Transfer Protocol

Consider the reconstruction of shared secret, $\llbracket a\rrbracket$ or $\langle\langle a\rangle\rangle$, over $\mathbb{N}$ as opposed to $\mathbb{Z}_p$ for which $\llbracket\cdot\rrbracket$ and $\langle\langle\cdot\rangle\rangle$ sharings are defined. The resulting value would be of the form $a + qp$. We will refer to q as the *quotient*. The ability to compute q (in shared form), which we refer to as quotient transfer, will play an important role in our division protocol.

Kikuchi *et al.* [28] proposed efficient three-party quotient transfer protocols for passive and active security. In this paper, we use their protocols specialized to the following setting; firstly, we use a Mersenne prime p for the field $\mathbb{Z}_p$ underlying the sharings, and we will use $\langle\langle\cdot\rangle\rangle$-sharing for passive security and $\llbracket\cdot\rrbracket$-sharing for active security. In this setting, a is required to be a multiple of 2 and 4 in the presence of passive and active adversaries, respectively. We address this by simply multiplying $\langle\langle a\rangle\rangle$ and $\llbracket a\rrbracket$ by 2 and 4, respectively, before conducting the quotient transfer protocols. It means that a secret a should satisfy $2a < p$ and $4a < p$ for passive and active security, respectively. These protocols have been implicitly used as building blocks for other protocols in [28], but were not explicitly defined. We thus describe them in Appendix B, as well as a quotient transfer protocol for $[\cdot]$-sharings, which is a popular way to obtain the carry for binary addition.

The quotient transfer functionality, $\mathcal{F}_{\text{QT}}$, is defined in Functionality 1. For brevity, $\mathcal{F}_{\text{QT}}$ is defined for $\langle\langle a\rangle\rangle$, but we additionally use this functionality for $\llbracket a\rrbracket$ and $[a]$. Note that for $\langle\langle a\rangle\rangle$, $q \in \{0,1\}$ where $\langle\langle a\rangle\rangle_1 + \langle\langle a\rangle\rangle_2 = a + qp$; for $\llbracket a\rrbracket$, $q \in \{0,1,2\}$ where sub-shares $a_1 + a_2 + a_3 = a + qp$; and for $[a]$, and $q \in \{0,1\}$ where sub-shares $a_1 + a_2 + a_3 = a + 2q$.

FUNCTIONALITY 1 ($\mathcal{F}_{QT}$ – Quotient transfer):
Upon receiving $\langle\langle a\rangle\rangle$, $\mathcal{F}_{\text{QT}}$ sets q such that $\langle\langle a\rangle\rangle_1 + \langle\langle a\rangle\rangle_2 = a + qp$ in $\mathbb{N}$, generates shares $\langle\langle q\rangle\rangle$, and sends $\langle\langle q\rangle\rangle_i$ to P_i.

III. Secure Real Number Operations

In this section, we present the division protocols that will allow us to do fixed-point arithmetic efficiently and securely. The key to achieve efficient fixed-point computations is the ability to perform *truncation* (or equivalently, integer division by 2^k, also called right-shift), as this allows multiplication of scaled integer representations of fixed-point values, as introduced in Section II-A.

Our new division protocol is efficient and accurate. The popular division (truncation) protocol approach used in the context of machine learning requires a heavy offline phase, although it is efficient in the online phase. In addition, as we will see later, there is a possibility of introducing errors that is much larger than rounding errors. Hence, we present a protocol that is efficient in its overall cost, i.e., the total cost is comparable only to the current *online* cost, while eliminating the possibility of introducing large errors.

A. Current Secure Division Protocol

In this section, we analyze the approach taken to division in current multi-party computation protocols [42], [56], [47] and show why the large error can be introduced in the output. For simplicity, we consider *unsigned* integers shared over $\mathbb{Z}_p$

for a general p and a general divisor d, but similar observations holds for the signed integers and specific p, such as 2^{64}.

Let (a_1, a_2, a_3) be the sub-shares of a in the replicated secret sharing scheme and $a = \alpha_a d + r_a$ for $0 \leq r_a < d$, and let (b_1, b_2, b_3) be the sub-shares of the output of a division protocol. Here, the intention is that $b_1 + b_2 + b_3$ is a value close to $\frac{a}{d}$, such as α_a, or perhaps $\alpha_a \pm 1$.

The typical protocol proceeds as follows. The parties first prepare a shared correlated randomness $(\llbracket s' \rrbracket, \llbracket s \rrbracket)$, where $s' \leftarrow \mathbb{Z}_p$ and $s := s'/d$. (Note that d is public and known a priori.) The parties then compute $\llbracket a + s' \rrbracket$, reconstruct $(a + s')$, and set $\llbracket b \rrbracket = \llbracket s \rrbracket + (a + s')/d$.

This protocol seems to work well, but the output can in fact be far from the intended $\frac{a}{d}$. To see this, let $s' = sd + r_{s'}$ and $p = \alpha_p d + r_p$. Considering the reconstructed value $\llbracket a + s' \rrbracket$ over $\mathbb{N}$, we see that the parties obtain $a + s' - qp$, where $q \in \{0, 1\}$. Hence, the computed shared secret corresponds to

$$\begin{aligned} & s + (a + s' - qp)/d \\ &= s + ((\alpha_a d + r_a) - (sd + r_{s'}) - q(\alpha_p d + r_p))/d \\ &= \alpha_a - q\alpha_p + (r_a - r_{s'} + qr_p)/d. \end{aligned} \tag{1}$$

Roughly speaking, the third term, $(r_a - r_{s'} - qr_p)/d$, is a small constant since r_a, $r_{s'}$, and r_p are less than d. Hence, this term can be considered to be a small rounding error. On the other hand, the second term, $q\alpha_p$, can be large if $q = 1$. For example, if we set $p = 2^{64}$ and $d = 2^{12}$, then $\alpha_p = 2^{52}$, and the reconstructed result will differ from $\frac{a}{d}$ by 2^{52}. To address this, we have to make the probability of $q = 1$ negligible. If a protocol conducts several divisions with $p = 2^k$ and ℓ-bit inputs, the probability that $q = 1$ occurs at least once during the τ divisions is $1 - \left(1 - 2^{-k+\ell}\right)^\tau$. It is likely to happen if we conduct the division protocol many times. For example, this probability is over 90% on 15 epochs MNIST training with the batch-size 128, $k = 64$, and $\ell = 36$. A single event of $q = 1$ does not necessarily lead to a catastrophic error in the intended function; however, many occurrences of $q = 1$ can make the result differ significantly from the intended value.

To keep the probability of an error occurring below a given threshold, the input space (i.e. the parameter ℓ) can be adjusted to be sufficiently small such that the required number of divisions can be accommodated. However, setting ℓ depending on the required number of divisions is often problematic, since this number can be difficult to estimate a priori in an exploratory analysis such as machine learning. Hence, a common approach is to set ℓ small enough to ensure $q = 0$ with overwhelming probability. This, however, leads to a larger reduction of the input space, which can negatively impact the computation being done, due to lower supported accuracy.

B. Our Protocol for Division by Public Value

1) Intuition: We first give the intuition behind our protocols. In our protocol for input $\llbracket a \rrbracket$, we locally convert $\llbracket a \rrbracket$ into $\langle\langle a \rangle\rangle$ before division. Hence, in the following, we assume the input is $\langle\langle a \rangle\rangle$ and a public divisor d.

First, let us analyze what happens when we simply divide each share by d. Let $\langle\langle a \rangle\rangle_1 + \langle\langle a \rangle\rangle_2 = qp + a$ in $\mathbb{N}$, where $q \in \{0, 1\}$. Here, suppose that $\langle\langle a \rangle\rangle_j = \alpha_j d + r_j$, $a = \alpha_a d + r_a$, and $p = \alpha_p d + r_p$ for $j = 1, 2$ If each party divide its share $\langle\langle a \rangle\rangle_j$ by d, the new share is α_j, *i.e.,* $\langle\langle a \rangle\rangle_j / d = \alpha_j$. Then, the reconstruction of (α_1, α_2) will be

$$\alpha_1 + \alpha_2 = \alpha_a + q\alpha_p + \frac{r_a + qr_p - (r_1 + r_2)}{d}, \tag{2}$$

which contains extra terms, $q\alpha_p$ and $\frac{(r_a + qr_p - (r_1 + r_2))}{d}$.

Protocol 1 Secure Division by Public Value in $\langle\langle \cdot \rangle\rangle$

Functionality: $\langle\langle c \rangle\rangle \leftarrow \mathsf{Div}_{(2,2)}(\langle\langle a \rangle\rangle, d)$
Input: Share of dividend $\langle\langle a \rangle\rangle$ and (public) divisor d, where a and d are even numbers.
Output: $\langle\langle c \rangle\rangle$, where $c \approx \frac{a}{d}$.
1: Let α_p and r_p be $p = \alpha_p d + r_p$, where $0 \leq r_p < d$.
2: $\langle\langle q \rangle\rangle \leftarrow \mathcal{F}_{\mathrm{QT}}(\langle\langle a \rangle\rangle)$
3: P_1 computes $\langle\langle b \rangle\rangle_1 \leftarrow (\langle\langle a \rangle\rangle_1 + d - 1 - r_p)/d$. ▷ "in $\mathbb{N}$" means no reduction $\mod p$
4: P_2 computes $\langle\langle b \rangle\rangle_2 \leftarrow \langle\langle a \rangle\rangle_2 / d$ in $\mathbb{N}$
5: $\langle\langle c \rangle\rangle := \langle\langle b \rangle\rangle - (\alpha_p + 1)\langle\langle q \rangle\rangle + 1$

Protocol 2 Secure Division by Public Value in $\llbracket \cdot \rrbracket$

Functionality: $\llbracket c \rrbracket \leftarrow \mathsf{Div}_{(2,3)}(\llbracket a \rrbracket, d)$
Input: Share of dividend $\llbracket a \rrbracket$ and public divisor d, where $0 \leq a \leq 2^{|p|-1} - 1$
Output: $\llbracket c \rrbracket$, where $c \approx \frac{a}{d}$.
1: $\langle\langle a \rangle\rangle \leftarrow \mathsf{ConvertToAdd}(\llbracket a \rrbracket)$
2: $\langle\langle a' \rangle\rangle := \langle\langle 2a \rangle\rangle, d' := 2d$
3: $\langle\langle c \rangle\rangle \leftarrow \mathsf{Div}_{(2,2)}(\langle\langle a' \rangle\rangle, d')$
4: $\llbracket c \rrbracket \leftarrow \mathsf{ConvertToRep}(\langle\langle c \rangle\rangle)$
5: Output $\llbracket c \rrbracket$

The insight behind our protocols is that the $q\alpha_p$ term can be eliminated, which we essentially achieve via the quotient transfer protocol [28], that allow us to obtain $\langle\langle q \rangle\rangle$ efficiently. This protocol suits our setting since it requires a prime p, and prefers a Mersenne prime. The quotient transfer protocol furthermore requires a to be a multiple of 2, but this is easily achieved by locally multiplying a and d with 2, and performing the division using $a' = 2a$ and $d' = 2d$. Note that the output of the division remains unchanged by this.

For the remaining error term $e = \frac{r_a + qr_p - (r_1 + r_2)}{d}$, each value r_a, r_p and r_j is less than d, and hence, $-1 \leq e \leq 2$. In our protocols, we reduce this error to $0 \leq e \leq 2$ by adding a combination of $\langle\langle q \rangle\rangle$ and appropriate constants to the output.

2) Passively Secure Protocols: We propose passively secure division protocols in Protocol 1 and 2. The first protocol works for input $\langle\langle a \rangle\rangle$, where a is a multiple of 2, and the second for $\llbracket a \rrbracket$ by extending the first protocol. We further extend our division protocols to *signed* integers a in Appendix D.

Both Protocol 1 and 2 have probabilistic rounding that outputs a/d, $a/d+1$, or $a/d+2$. In other words, our protocols guarantees that there is only a small difference between $\frac{a}{d}$ and the output of our protocols, while the standard approach to division protocols have a similar rounding difference *and* a large error $q\alpha_p$ with a certain probability. The functionalities of these protocols appear in Appendix C. Note, the most interesting case in which p is a Mersenne prime and d is a power of 2, the protocol outputs either a/d or $a/d + 1$.

3) Output of Protocols: Consider the reconstruction of the output shares calculated returned by Protocol 1. We obtain (see

Section III-B1 for notation):

$$
\begin{aligned}
&(\langle\!\langle b \rangle\!\rangle_1 - (\alpha_p + 1)\langle\!\langle q \rangle\!\rangle_1 + 1) + (\langle\!\langle b \rangle\!\rangle_2 - (\alpha_p + 1)\langle\!\langle q \rangle\!\rangle_2) \\
&= \langle\!\langle b \rangle\!\rangle_1 + \langle\!\langle b \rangle\!\rangle_2 - (\alpha_p + 1)(\langle\!\langle q \rangle\!\rangle_1 + \langle\!\langle q \rangle\!\rangle_2) + 1 \\
&= (\langle\!\langle a \rangle\!\rangle_1 + d - 1 - r_p)/d + \langle\!\langle a \rangle\!\rangle_2/d - (\alpha_p + 1)q + 1 \\
&= (\alpha_1 d + r_1 + d - 1 - r_p)/d + (\alpha_2 d + r_2)/d - q\alpha_p - q + 1 \\
&= \alpha_1 + \alpha_2 - q\alpha_p - q + 1 + (r_1 - r_p + d - 1)/d + r_2/d
\end{aligned}
$$

From Eq. (2) and the fact that $r_1 \leq d - 1$, the last equation equals to

$$\alpha_a - q + 1 + \frac{r_a + qr_p - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d. \quad (3)$$

The above computation shows the term $q\alpha_p$ being eliminated. The remaining terms are quite small since $q \in \{0, 1\}$, each r_i is smaller than d, and $\frac{r_a + qr_p - r_1 - r_2}{d}$ and $(r_1 - r_p + d - 1)/d$ are at most 1. To see that the output in Eq. (3) corresponds to the output defined in the ideal functionalities defined in Appendix C, a more precise analysis is required. In Appendix E we provide a detailed analysis of both the simpler specific case of p being a Mersenne prime and d a power of 2, and the general case. Note that this analysis is required to formally establish the security of our protocols.

Lastly, note that the distribution of the output expressed in Eq. (3), depends on the value of the input *shares* (as opposed to the reconstructed value). We provide a full analysis of the output distribution in Appendix F. The output range and distribution of Protocol 2 follows that of Protocol 1.

4) Security: The following theorem establishes security of Protocol 1.

Theorem 2: Protocol 1 securely computes the division functionality $\mathcal{F}_{\text{div}}$ in the $\mathcal{F}_{\text{QT}}$-hybrid model in the presence of a passive adversary.

The proof of this immediately follows from the correctness discussion in above, since Protocol 1 only consists of local computations except for $\mathcal{F}_{\text{QT}}$.

The security of Protocol 2 follows from that of Protocol 1. Here, ConvertToRep is the only additional step requiring communication compared to Protocol 1. This step can easily be simulated since the output consists only of shares, and the simulator can simply insert a random share as to simulate transcripts.

5) Efficiency: We obtain the concrete efficiency of our protocols by considering efficient instantiations of the required building blocks.

The quotient transfer protocol in [28] requires 2 bits of communication and 1 communication round, besides a single call of $\mathcal{F}_{\text{mod}}$. The modulus conversion protocol in [28] and ConvertToRep require $3|p| + 3$ bits and $2|p|$ bits of communication, respectively, and both 1 communication round. Furthermore, we can reduce the number of rounds required in Protocol 2 by parallel execution[2] of $\mathcal{F}_{\text{QT}}$ and ConvertToRep. Consequently, instantiating $\mathcal{F}_{\text{QT}}$ (and $\mathcal{F}_{\text{mod}}$ used in QT internally) by the protocols in [28], Protocol 1 and 2 require $3|p|+5$ and $5|p| + 5$ bits of communication and 2 communications rounds in total.

[2]We change the protocol by skipping ConvertToAdd in step 4 in Protocol 10 (hence the output of $\mathcal{F}_{\text{QT}}$ is $[\![q]\!]$) and local computation in step 5 in Protocol 1 is performed over $[\![\cdot]\!]$-sharings by computing ConvertToRep($\langle\!\langle b \rangle\!\rangle$) just before this step.

6) Comparison: We compare Protocol 2 with the one (denoted truncation in the original paper) of ABY3 [42], which has been used in subsequent literature, such as FALCON. Since the ABY3 protocol is proposed as truncation, we let $d = 2^\delta$. The ABY3 protocol requires $6(2|p| - \delta - 1)$ bits and $|p| - 1$ rounds in the offline phase, and $3|p|$ bits and 1 round in the online phase, so the total communication cost is $15|p| - 6\delta - 6$ bits and $|p|$ rounds. In comparison, our protocol requires $5|p| + 5$ bits and 2 rounds. Thus, the total cost of our protocol is much better compared to ABY3's, and slightly worse than ABY3's *online* cost. In addition, the ABY3's protocol can contain a large error $\frac{p}{d}$ with a certain probability. This leads to a small message spaces and/or less accuracy, as pointed out in Sec. III-A.

7) Active Security: In the above, we have only treated passively secure protocols. We next outline how we construct a division protocol satisfying active security with abort. A quotient transfer protocol secure against an active adversary with abort has already been proposed in [28]. We can thus employ the same approach as in the passive security: eliminating $q\alpha_p$ via the use of $\mathcal{F}_{\text{QT}}$.

The difference between our actively and passively secure division protocols, is that the only known efficient actively secure quotient transfer protocol uses input $[\![a]\!]$, where a is required to be a multiple of 4. Hence, our actively secure division protocol works directly on $[\![a]\!]$ and essentially executes the following steps:

1) $[\![a']\!] := 4[\![a]\!]$ and $d' := 4d$
2) $[\![q]\!] \leftarrow \mathcal{F}_{\text{QT}}([\![a]\!])$
3) Parties divide own sub-shares of $[\![a']\!]$ by d', and let them be $[\![b]\!]$.
4) $[\![c]\!] := [\![b]\!] - \alpha_p[\![q]\!]$

In addition, we further adjust the output using appropriate constants to minimize the difference between $\frac{a}{d}$ and the protocol output. The concrete protocol appears in Appendix G.

Regarding security, as we do not convert to $\langle\!\langle a \rangle\!\rangle$, the quotient transfer protocol is the only step that requires communication, and the security of the protocol is thus trivially reduced to $\mathcal{F}_{\text{QT}}$.

IV. Elementary functions for machine learning

In the following, we will present efficient and high-accuracy protocols for arithmetic functions suitable for machine learning, such as inversion, square root extraction, and exponential function evaluation. All of these rely on fixed-point arithmetic, and will make use of the division protocols introduced above to implement this. Recall that, the product of fixed-point values a and b with offset ℓ, written $a \times_\ell b$, corresponds to (standard) multiplication of a and b, followed by division by 2^ℓ. To ease notation in the protocols, we use $[\![a]\!] \cdot [\![b]\!]$ to denote $\mathcal{F}_{\text{mult}}([\![a]\!], [\![b]\!])$, and $[\![a]\!] \times_\ell [\![b]\!]$ to denote $\mathcal{F}_{\text{div}}([\![a]\!] \cdot [\![b]\!], 2^\ell)$.

All protocols will explicitly have as parameters the offset of both input and output values, often denoted α and δ, and in particular will allow these to be different. This can be exploited to obtain more accurate computations when reasonable bounds for the input and output are publicly known. For example,

consider the softmax function, often used in neural networks, defined as $\frac{e^{u_i}}{\sum_{j=0}^{k-1} e^{u_j}} = \frac{1}{\sum_{j=0}^{k-1} e^{u_j - u_i}}$ for input $(u_0, \ldots, u_{k-1})$. The output is a value between 0 and 1, and to maintain high accuracy, the offset should be large, e.g. 23 to maintain 23 bits of accuracy below the decimal point. However, the computation of $\sum_{j=0}^{k-1} e^{u_j - u_i}$ is often expected to be a large value in comparison, and a much smaller offset can be used to prevent overflow e.g. -4. Furthermore, we highlight that internally, some of the protocols will switch to using an offset different from α and δ to obtain more accurate numerical computations. By fine-tuning and tailoring the offsets to the computations being done, the most accurate computation with the available ℓ bits for shared values, can be obtained.

We will first show protocols for unsigned inputs, and defer the extension to signed inputs to Sect. IV-E.

A. Inversion

In the following, we introduce a protocol for computing the inverse of a shared fixed-point value. This is a basic operation required for many computations, including machine learning.

Before presenting the inversion protocol itself, we introduce a specialized bit-level functionality of *private scaling* that will compute a representation of the input $[\![a]\!]$ which allows us to make full use of the available bit range for shared values. Specifically, the representation of $[\![a]\!]$ is $[\![b]\!] = [\![a]\!] \cdot [\![c]\!]$, where $2^{\ell-1} \leq b \leq 2^\ell - 1$ and c is a power of 2 (recall that shared values are ℓ bit integers). This functionality corresponds to a left-shift of the shared value $[\![a]\!]$ such that the most significant *non-zero* bit becomes the most significant bit, where c represents the required shift to obtain this. We will denote this operation $\mathsf{MSNZBFit}$ (MSNZB denoting Most Significant Non-Zero Bit) and the corresponding functionality $\mathcal{F}_{\text{msnzbfit}}$. The protocol presented in Protocol 3 implements this functionality. Recall that $\mathcal{F}_{\text{BC}}$ and $\mathcal{F}_{\text{BDC}}$ are the functionalities of bit-(de)composition.

Theorem 3: Protocol 3 securely implements $\mathcal{F}_{\text{msnzbfit}}$ in the ($\mathcal{F}_{\text{BDC}}$, $\mathcal{F}_{\text{BC}}$, $\mathcal{F}_{\text{mult}}$)-hybrid model in the presence of a passive adversaries.

Protocol 3 MSNZB Fitting

Functionality: $([\![b]\!], [\![c]\!]) \leftarrow \mathsf{MSNZBFit}([\![a]\!])$
Input: $[\![a]\!]$
Output: $[\![b]\!], [\![c]\!]$, where $[\![b]\!] = [\![a]\!] \cdot [\![c]\!]$, $2^{\ell-1} \leq b \leq 2^\ell - 1$, and $c = 2^e$ for some $e \in \mathbb{N}$.
Parameter: ℓ

1: $([a_1], \ldots, [a_\ell]) \leftarrow \mathcal{F}_{\text{BDC}}([\![a]\!])$
2: $[f_\ell] := [a_\ell]$
3: **for** $i = \ell - 1$ to 1 **do**
4: $\quad [f_i] := [f_{i+1}] \vee [a_i]$ ▷ $f_i = 1$ for all i corresponding to MSNZB of a or smaller
5: $[x_\ell] := [a_\ell]$
6: **for** $i = \ell - 1$ to 1 **do**
7: $\quad [x_i] := [f_i] \oplus [f_{i+1}]$ ▷ $x_i = 1$ only for i corresponding to MSNZB of a
8: $[\![c]\!] \leftarrow \mathcal{F}_{\text{BC}}([x_\ell], \ldots, [x_1])$ ▷ Bit-compose $[x_i]$ *in the reverse order* to obtain $c = 2^{\ell-1-\lfloor \log_2 a \rfloor}$
9: $[\![b]\!] = [\![a]\!] \cdot [\![c]\!]$
10: Output $[\![b]\!]$ and $[\![c]\!]$

Protocol 4 Inversion

Functionality: $[\![d]\!] \leftarrow \mathsf{Inv}([\![a]\!])$
Input: $[\![a]\!]$, where $a \in \widehat{\mathbb{Q}}_{\langle \ell, \alpha \rangle}$
Output: $[\![d]\!]$, where $d \approx \left(\frac{1}{a}\right)_{\langle \delta \rangle}$
Parameter: $(\ell, I, \alpha, \delta)$, where I is the number of iterations (say, $I = \lceil \log \ell \rceil$) used in the computation

1: $([\![b]\!], [\![c]\!]) \leftarrow \mathsf{MSNZBFit}([\![a]\!])$ ▷ $b = b' \cdot 2^\ell$ where $b' \in [\frac{1}{2}, 1)$
2: $[\![x_1]\!] := 1_{\langle \ell \rangle} - [\![b]\!]$
3: $[\![y_1]\!] := 2_{\langle \ell \rangle} - [\![b]\!]$
4: **for** $i = 2$ to I **do**
5: $\quad [\![x_i]\!] := [\![x_{i-1}]\!] \times_\ell [\![x_{i-1}]\!]$
6: $\quad [\![y_i]\!] := [\![y_{i-1}]\!] \times_\ell (1_{\langle \ell \rangle} + [\![x_{i-1}]\!])$
7: Output $[\![y_I]\!] \cdot [\![c]\!] \cdot 2^{\alpha+\delta-2\ell}$

Using $\mathsf{MSNZBFit}$ as a building block, we now construct our inversion protocol. The protocol is based on the Taylor series for $(1 - x_0)^{-1}$ centered around 0, where $x \in [0; \frac{1}{2})$:

$$\frac{1}{1 - x_1} = \sum_{i=0}^{\infty} x_1^i = 1 + x_1 + x_1^2 + \cdots . \tag{4}$$

Continuing this Taylor series until the n-degree, yields the remainder term $\frac{x^{n+1}}{1-x} \leq \frac{1}{2^n}$, which implies that the approximation has n bits accuracy.

Firstly, we use $\mathsf{MSNZBFit}$ to left-shift input $[\![a]\!]$ to obtain $[\![b]\!] = [\![a]\!] \cdot [\![c]\!]$. Interpreting the resulting value $[\![b]\!]$ as being a fixed-point value with offset ℓ implies that $b \in [\frac{1}{2}; 1)$. This representation forms the basis of our computation. (Note that since $b \in [\frac{1}{2}; 1)$, we have that $\frac{1}{b} \leq 2$, ensuring that $\frac{1}{b}$ can be represented using $\ell + 1$ bits.)

For the computation of $\frac{1}{b}$, instead of using Eq. (4) directly, which requires r multiplications for $(r+1)$-th degree terms, we use the following product requiring only $\log r$ multiplications.

$$\prod_{j=0}^{\infty} (1 + x_1^{2^j}) = (1 + x_1)(1 + x_1^2)(1 + x_1^4) \cdots . \tag{5}$$

Letting $b = 1 - x_1$ (which ensures $x_1 \in [0; \frac{1}{2})$), our inversion protocol shown in Protocol 4 iteratively computes Eq. (5) by first setting $x_1 = 1 - b$ and $y_1 = 1 + x_1 = 2 - b$ (Step 2-3), and in each iteration computing $(1 + x_1^i)$ and multiplying this with y_i (Step 4-6). The number of iterations is specified via the parameter I. Finally, to obtain (an approximation to) $[\![\frac{1}{a}]\!]$, we essentially only need to scale the computed $[\![y_I]\!] = [\![\frac{1}{b}]\!]$ with $[\![c]\!]$ (as $\frac{1}{b} \cdot c = \frac{1}{ac} \cdot c = \frac{1}{a}$). Note, however, that the output has to be scaled taking into account the input and output offsets, as well as the offset used in the internal computation. To see that the correct scaling factor is $2^{\alpha+\delta-2\ell}$, note that for input $a = a'_{\langle \alpha \rangle}$ and output $b = a \cdot c = b'_{\langle \ell \rangle}$ of $\mathsf{MSNZBFit}$, we have

$$y_I = \frac{2^\ell}{b'} = \frac{2^\ell}{a' \cdot 2^\alpha \cdot c \cdot 2^{-\ell}} = \frac{1}{a' \cdot c} \cdot 2^{2\ell - \alpha}$$

and that the output should be scaled with 2^δ.

We define the corresponding functionality $\mathcal{F}_{\text{Inv}}$ in which on input of shares computes the above Taylor series expansion and output shares of that output.

Theorem 4: The protocol Inv securely computes inversion functionality $\mathcal{F}_{\text{Inv}}$ in the ($\mathcal{F}_{\text{msnzbfit}}, \mathcal{F}_{\text{div}}, \mathcal{F}_{\text{mult}}$)-hybrid model in the presence of a passive adversary.

Protocol 5 Integer Division with Private Divisor

Functionality: $[\![z]\!] \leftarrow \mathsf{Div}_{\mathsf{priv}}([\![a]\!], [\![d]\!])$
Input: $[\![a]\!], [\![d]\!]$, where $a = a'_{\langle\alpha\rangle} \in \mathbb{Q}_{\langle\ell,\alpha\rangle}$, and $d = d'_{\langle\beta\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\beta\rangle}$
Output: $[\![z]\!]$, where $z \approx \left(\frac{a'}{d'}\right)_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \beta, \delta)$ where α and β are the offsets of a and d, respectively.
1: $[\![z']\!] \leftarrow \mathsf{Inv}_{(\ell,I,\beta,\delta)}([\![d]\!])$ ▷ $z' = \left(\frac{1}{d'}\right)_{\langle\delta\rangle}$
2: Output $[\![z']\!] \cdot [\![a]\!] \cdot 2^{-\alpha}$

B. Division with Private Divisor

Given our protocol for inversion, it becomes trivial to construct a high accuracy protocol for division with a private divisor. Specifically, for values $[\![a]\!]$ and $[\![d]\!]$, we simply compute $[\![\frac{1}{d}]\!]$ using Inv, and multiply this with $[\![a]\!]$ to obtain $[\![\frac{a}{d}]\!]$. The resulting protocol, $\mathsf{Div}_{\mathsf{priv}}$, is shown in Protocol 5. Note that the accuracy of the result is determined by the parameter I of the inversion protocol. Setting $I = \log \ell$ gives ℓ bits of precision for the inversion, which ensures the result is equal to $\frac{a}{d}$ for ℓ-bit fixed-point values. We define the corresponding functionality $\mathcal{F}_{\text{divpriv}}$ that as on input $[\![a]\!]$ and $[\![d]\!]$ outputs $[\![\frac{a}{d}]\!]$ in which $[\![\frac{1}{d}]\!]$ is obtained by $\mathcal{F}_{\text{Inv}}$.

Theorem 5: The protocol $\mathsf{Div}_{\mathsf{priv}}$ securely computes fixed-point division $\mathcal{F}_{\text{divpriv}}$ in the $(\mathcal{F}_{\text{Inv}}, \mathcal{F}_{\text{div}}, \mathcal{F}_{\text{mult}})$-hybrid model in the presence of a passive adversary.

C. Square Root and Inverse Square Root

Computing the inverse of the square root of an input value, is a useful operation for many computations, *e.g.,* normalization of a vector, and is likewise used in Adam. Hence, having an efficient protocol for directly computing this, is beneficial.

Our protocol for computing the inverse of a square root is shown in Protocol 7, and is based on Newton's method for the function $f(y) = \frac{1}{y^2} - x$ for input value x (note that $f(y') = 0$ implies $y' = \frac{1}{\sqrt{x}}$). This involves iteratively computing approximations

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = \frac{y_n(3 - x \cdot y^2)}{2}$$

for an appropriate initial guess y_0 (Step 4-5 performs this iteration). To ensure fast convergence for a large range of input values, we represent the input $x = b \cdot 2^e$ for $b \in [\frac{1}{2}; 1)$, which implies

$$\frac{1}{\sqrt{x}} = \begin{cases} \left(\frac{1}{\sqrt{b}}\right) \cdot 2^{-e/2} & \text{if } e \text{ is even} \\ \left(\frac{\sqrt{2}}{\sqrt{b}}\right) \cdot 2^{-(e+1)/2} & \text{if } e \text{ is odd} \end{cases}.$$

Hence, we only need to compute $\frac{1}{\sqrt{b}}$ for $b \in [\frac{1}{2}; 1)$, in which case using 1 as the initial guess provides fast convergence. However, the parties should not learn which of the above two cases the input falls into. We introduce a sub-protocol, $\mathsf{MSNZBFitExt}$ shown in Protocol 6, that computes values r and c', where $(r, c') = (0, 2^{e/2})$ if x falls into the first case, and $(r, c') = (1, 2^{(e+1)/2})$ otherwise. Note that like $\mathsf{MSNZBFit}$, the extended $\mathsf{MSNZBFitExt}$ right-shifts the input a to obtain an ℓ-bit value $b = a \cdot c$, that when interpreted as an element of $\widehat{\mathbb{Q}}_{\langle\ell,\ell\rangle}$, represents $b \in [\frac{1}{2}; 1)$, and that $c' = \sqrt{c}$. Having r and c' allows us to compute $\frac{1}{\sqrt{x}}$ as $\frac{1}{\sqrt{b}} \cdot (1 + r \cdot (\sqrt{2} - 1)) \cdot c'$. Finally, note

Protocol 6 $\mathsf{MSNZBFitExt}$ Sub-protocol for $\mathsf{InvSqrt}$

Functionality: $([\![b]\!], [\![c']\!], [\![r]\!]) \leftarrow \mathsf{MSNZBFitExt}([\![a]\!])$
Input: $[\![a]\!]$
Output: $([\![b]\!], [\![c']\!], [\![r]\!])$, where $b = b'_{\langle\ell\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\ell\rangle}$ and $b' \in [\frac{1}{2}; 1)$, $x = b' \cdot 2^e$, and $r = 0$ if e is even, and $r = 1$ otherwise.
Parameter: ℓ
1: Parties jointly execute steps 1-9 of protocol $\mathsf{MSNZBFit}$.
2: $\ell' := \lfloor \frac{\ell}{2} \rfloor$
3: $[x'_i] := [x_{\ell+1-i}]$ for $1 \le i \le \ell$
4: **for** $i = 1$ to $\ell' - 1$ **do**
5: $\quad [y_i] := [x'_{2i}] \oplus [x'_{2i+1}]$
6: **if** ℓ is an even number **then**
7: $\quad [y_{\ell'}] := [x'_{2\ell'}] \oplus [x'_{2\ell'+1}]$
8: **else**
9: $\quad [y_{\ell'}] := [x'_{2\ell'}]$
10: $[r] := [x'_2] \oplus [x'_4] \oplus \cdots \oplus [x'_{2\lfloor \frac{\ell'}{2} \rfloor}]$
11: $[\![r]\!] \leftarrow \mathcal{F}_{\text{mod}}([r])$
12: $[\![c']\!] \leftarrow \mathcal{F}_{\text{BC}}([y_1], \ldots, [y_{\ell'}])$
13: Output $([\![b]\!], [\![c']\!], [\![r]\!])$

Protocol 7 Inversion of Square Root

Functionality: $[\![z]\!] \leftarrow \mathsf{InvSqrt}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle\alpha\rangle} \in \mathbb{Q}_{\langle\ell,\alpha\rangle}$
Output: $[\![z]\!]$, where $z \approx \left(\frac{1}{\sqrt{a'}}\right)_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \delta)$, where I is the number of iteration (say, $I = \lceil \log \ell \rceil$) in the computation.
1: $([\![b]\!], [\![c']\!], [\![r]\!]) \leftarrow \mathsf{MSNZBFitExt}([\![a]\!])$
2: $[\![x_1]\!] := 3_{\langle\ell\rangle} - [\![b]\!]$
3: $[\![y_1]\!] := [\![x_1]\!]/2$
4: **for** $i = 2$ to I **do**
5: $\quad [\![x_i]\!] := 3_{\langle\ell\rangle} - ([\![y_{i-1}]\!] \times_\ell [\![y_{i-1}]\!]) \cdot [\![b]\!]$
6: $\quad [\![y_i]\!] := [\![x_{i-1}]\!] \times_{\ell+1} [\![y_{i-1}]\!]$ ▷ Implicit scaling by $\frac{1}{2}$
7: Output $[\![y_I]\!] \cdot (1 + [\![r]\!] \cdot (\sqrt{2} - 1)) \cdot [\![c']\!] \cdot 2^{\delta - \frac{3}{2}\ell + \frac{\alpha}{2}}$

that the output has to be scaled, taking into account the input and output offsets, as well as the offset used for the internal computation. To see that the correct scaling factor is $2^{\delta - \frac{3}{2}\ell + \frac{\alpha}{2}}$, note that for input $a = a'_{\langle\alpha\rangle}$ and output $b = a \cdot (c')^2 = b' * 2^\ell$ of $\mathsf{MSNZBFitExt}$, we have

$$y_I \approx \frac{2^\ell}{\sqrt{b'}} = \frac{2^\ell}{\sqrt{a' \cdot 2^\alpha \cdot (c')^2 \cdot 2^{-\ell}}} = \frac{1}{\sqrt{a'}} \cdot \frac{1}{c'} \cdot 2^{\ell/2 - \alpha/2}$$

and that the output should be scaled with 2^δ. We define the functionality $\mathcal{F}_{\text{InvSqrt}}$ that computes $\frac{1}{\sqrt{x}}$ as done in the above using the Newton's method.

Theorem 6: The protocol $\mathsf{InvSqrt}$ securely computes the inverse of the square root functionality $\mathcal{F}_{\text{InvSqrt}}$ in the $(\mathcal{F}_{\text{msnzbfit}}, \mathcal{F}_{\text{mod}}, \mathcal{F}_{\text{div}}, \mathcal{F}_{\text{mult}})$-hybrid model in the presence of a passive adversary.

Given the above protocol $\mathsf{InvSqrt}$ for computing $\frac{1}{\sqrt{x}}$, and noting that $\sqrt{x} = \frac{x}{\sqrt{x}}$, we can easily construct a protocol for computing $\sqrt{x}$, simply by running $\mathsf{InvSqrt}$ and multiplying the result with x. The resulting protocol, Sqrt, is shown in Protocol 8. Let $\mathcal{F}_{\text{sqrt}}$ be the functionality that on input $[\![a]\!]$ outputs $[\![\sqrt{a}]\!]$ in which $[\![\frac{1}{\sqrt{a}}]\!]$ is obtained by $\mathcal{F}_{\text{InvSqrt}}$.

Theorem 7: The protocol Sqrt securely computes the square root functionality $\mathcal{F}_{\text{sqrt}}$ in the $(\mathcal{F}_{\text{InvSqrt}}, \mathcal{F}_{\text{div}}, \mathcal{F}_{\text{mult}})$-

Protocol 8 Square Root

Functionality: $[\![z]\!] \leftarrow \mathsf{Sqrt}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle\alpha\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\alpha\rangle}$
Output: $[\![z]\!]$, where $z \approx (\sqrt{a'})_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \delta)$ where I is the number of iterations used in the computation.

1: $[\![z']\!] \leftarrow \mathsf{InvSqrt}_{(\ell,I,\alpha,\delta)}([\![a]\!])$ ▷ $z' = \left(\frac{1}{\sqrt{a'}}\right)_{\langle\delta\rangle}$
2: Output $[\![a]\!] \cdot [\![z]\!] \cdot 2^{-\alpha}$

hybrid model in the presence of a passive adversary.

D. Exponential Function

To obtain a fast and highly accurate protocol for evaluating the exponential function, we adopt what we call *hybrid table-lookup/series-expansion* technique. Intuitively, it utilizes the table lookup approach for the large-value part of the input, in combination with the Taylor series evaluation for its small-value counterpart. We first recall that the Taylor series of the exponential function is

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots .$$

which converges fast for small values of x. To minimize the value for which we use the above Taylor series, we separate the input a into three parts:

1) μ: a lower bound for the input
2) $b_\ell, \ldots, b_{\ell-t}$: bit representation of the t most significant bits of $b := a - \mu$
3) b_σ: integer representation of $(a-\mu) - \sum_{\ell-t \le i \le \ell} 2^{i-\alpha} b_i$

which means that we can compute $\exp(a)$ as

$$\exp a = \left(\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^{i-\alpha}) \right) \cdot \exp(b_\sigma) \cdot \exp(\mu). \tag{6}$$

Here, α is the input offset and t is a parameter of our protocol that determines which part of the input we will evaluate using table lookups, and which part we will evaluate using a Taylor series. In this product, we evaluate $\exp(\mu)$ locally (as μ is public), $\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^i)$ using table lookups, and $\exp(b_\sigma)$ using the Taylor series. The taylor series rapidly converge since b_σ is made small due to the subtraction of μ and the value of the t most significant bits of a.

More specifically, for the table lookup computation, note that the binary value b_i determines whether the factor $\exp 2^i$ will included. Hence, by combining bit decomposition, that allows parties to compute $[b_i]$ from $[\![b]\!]$, with the CondAssign protocol using $[b_i]$ as the condition, and the values 1 and $\exp 2^i$, which are public and can be precomputed, we obtain an efficient mechanism for computing $\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^i)$. However, to maintain high accuracy, the parties will not use $\exp 2^i$ directly, but precompute a mantissa f_i and exponent 2^{ϵ_i} such that $f_i \cdot 2^{\epsilon_i} = \exp 2^{i-\alpha}$. This allows the parties to compute the product of f_i values separately from the product of 2^{ϵ_i} values, and only combine these in the final step constructing the output, thereby avoiding many of the rounding errors that potentially occur in large products of increasingly larger values.

Protocol 9 Exponential Function

Functionality: $[\![z]\!] \leftarrow \mathsf{Exponent}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle\alpha\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\alpha\rangle}$
Output: $[\![z]\!]$, where $z \approx (\exp a')_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \beta, \delta, \mu, t)$ where I is the number of iterations used in the computation, β is the offset of the lookup table values, t indicates the lookup table vs Taylor series threshold, and μ is a lower bound for the input.

1: $[\![b]\!] := [\![a]\!] - \mu_{\langle\alpha\rangle}$
2: $[b_\ell], \ldots, [b_{\ell-t}] \leftarrow \mathcal{F}_{\text{BDC}}([\![b]\!])$ ▷ We use only $\ell - t$ MSBs while $\mathcal{F}_{\text{BDC}}$ outputs ℓ bits.
3: $[\![b_i]\!] \leftarrow \mathcal{F}_{\text{mod}}([b_i])$ for $i = \ell, \ldots \ell - t$.
4: $[\![b_\sigma]\!] := [\![b]\!] - \sum_{\ell-t \le i \le \ell} 2^i [\![b_i]\!]$ ▷ Value of t LSBs of $[\![b]\!]$
5: Parties define f_i, ϵ_i such that $\exp 2^{i-\alpha} = f_i \cdot 2^{\epsilon_i}$ ▷ Precomputed lookup table values
6: Using $[\![f'_i]\!] \leftarrow \mathcal{F}_{\text{CondAssign}}(1_{\langle\beta\rangle}, (f_i)_{\langle\beta\rangle}, [b_i])$, the parties obtain $[\![f'_i]\!] = \begin{cases} (f_i)_{\langle\beta\rangle} & \text{if } b_i = 1 \\ 1_{\langle\beta\rangle} & \text{otherwise} \end{cases}$ for $i = \ell, \ldots, \ell - t$
7: Using $[\![\epsilon'_i]\!] \leftarrow \mathcal{F}_{\text{CondAssign}}(1, 2^{\epsilon_i}, [b_i])$, the parties obtain $[\![\epsilon'_i]\!] = \begin{cases} 2^{\epsilon_i} & \text{if } b_i = 1 \\ 1 & \text{otherwise} \end{cases}$ for $i = \ell, \ldots, \ell - t$
8: $[\![f']\!] := [\![f'_\ell]\!] \times_\beta \ldots \times_\beta [\![f'_{\ell-t}]\!]$
9: $[\![\epsilon']\!] := [\![\epsilon'_\ell]\!] \cdot \ldots \cdot [\![\epsilon'_{\ell-t}]\!]$
10: $b_{\sigma,0} := 1$
11: **for** $i = 1$ to $I - 1$ **do**
12: $\quad [\![b_{\sigma,i}]\!] \leftarrow [\![b_{\sigma,i-1}]\!] \times_\alpha [\![b_\sigma]\!]$
13: $[\![b'_\sigma]\!] := \sum_{0 \le i < I} \frac{[\![b_{\sigma,i}]\!]}{i!}$ ▷ Division using Div
14: Output $[\![f']\!] \cdot [\![\epsilon']\!] \cdot [\![b'_\sigma]\!] \cdot \exp\mu \cdot 2^{\delta-\beta-\alpha}$

Lastly, the result is computed as $\prod_i f_i \cdot \prod_i 2^{\epsilon_i} \cdot \exp a_\sigma$. Note that the input and output offsets have to be taken into account, and the output adjusted appropriately. We define the functionality $\mathcal{F}_{\text{exp}}$ such that on input $[\![a]\!]$, e^a is computed as done in the above and output $[\![e^a]\!]$.

Theorem 8: The protocol Exponent securely computes the exponential function functionality $\mathcal{F}_{\text{exp}}$ in the $(\mathcal{F}_{\text{CondAssign}}, \mathcal{F}_{\text{mod}}, \mathcal{F}_{\text{BDC}}, \mathcal{F}_{\text{div}}, \mathcal{F}_{\text{mult}})$-hybrid model in the presence of a passive adversary.

E. Extension to signed integer

We extend the protocols proposed in this section into those for signed integers. We generically convert the protocols by extracting sign and absolute of an input at first. The protocol that extract sign and absolute appears in Appendix H. Obtaining sign and absolute, we conduct the protocols on the absolute, and then multiply the sign to obtain the output of signed integer.

F. Satisfying Active Security with Abort

There are known compilers that convert a passively secure protocol to an actively secure one (with abort). The compiler [12] and its extension [27] can be applied to Binary/arithmetic circuit computation, and each step of our proposed protocols except $\mathcal{F}_{\text{BDC}}$, $\mathcal{F}_{\text{mod}}$, and $\mathcal{F}_{\text{div}}$ is circuit computation over modulus 2 and p. Therefore, we can obtain actively secure versions of our protocols computing elementary functions by applying that compiler on modulus 2 and p in parallel.

V. Secure Deep Neural Network

The main application we consider in this paper for our secure protocols, is the construction of secure deep neural networks. We will first give a brief overview of the functions required to implement deep neural networks, and then present secure protocols for these. In particular, we propose efficient secure protocols for the softmax activation function and the Adam optimization algorithm for training.

A. Neural Network

In this paper, we deal with feedforward and convolutional neural networks. A network with two or more hidden layers is called a deep neural network, and learning in such a network is called deep learning.

A layer contains neurons and the strength of the coupling of neurons between adjacent layers described by parameters w_i. Learning is a process that (iteratively) updates the parameters to obtain the appropriate output.

1) Layer: There are several types of layers. The fully connected layer is computing the inner product of the input vector with the parameters. The convolutional layer is a fully-connected layer with removing some parameters and computations. The max-pooling computes the max value in an input vector to obtain a representative. The batch normalization performs normalization and an affine transformation of the input. To normalize a vector $\overrightarrow{x} = (x_1, \ldots, x_n)$, we compute

$$x_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{7}$$

where μ and σ are mean and variance of $\overrightarrow{x}$, and ϵ is a small constant.

2) Activation Function: In a neural network, the activation functions of the hidden layer and the output layer are selected according to purpose.

ReLU Function A popular activation function at the middle layer is the ReLU function defined as $\mathbf{ReLU}(u) = \max(0, u)$. the function $\mathbf{ReLU}'(u)$, outputting 0 if $u \leq 0$ and 1 otherwise, is used as (a substitute of) differentiated ReLU function.

Softmax Function Classifications for image identification commonly use the softmax function $\mathbf{softmax}(u_i)$ at the output layer. The softmax function for classification into k classes is as follows:

$$\mathbf{softmax}(u_i) = \frac{e^{u_i}}{\sum_{j=0}^{k-1} e^{u_j}} = \frac{1}{\sum_{j=0}^{k-1} e^{u_j - u_i}}. \tag{8}$$

3) Optimization: A basic method of parameter update is stochastic gradient descent (SGD). This method is relatively easy to implement but has drawbacks such as slow convergence and the potential for becoming stuck at local maxima. To address these drawbacks, optimized algorithm have been introduced. [52] analyzed eight representative algorithms, and Adam [29] was found to be providing particularly good performance. In fact, Adam is now used in several machine learning framework [26], [53].

The process of Adam includes the equation

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{V}_{t+1}} + \epsilon} \circ \hat{M}_{t+1}, \tag{9}$$

TABLE III: Environment

OS	CentOS Linux release 7.3.1611
CPU	Intel Xeon Gold 6144k (3.50GHz 8 core/ 16 thread) × 2
Memory	768 GB
NW	Intel X710/X557-AT 10G Ring configuration

where t indicates the iteration number of the learning process, $W, \hat{V}$, and $\hat{M}$ are matrices, $\circ$ denotes the element-wise multiplication, and η and ϵ are parameters.

B. Secure Protocols for Deep Neural Networks

The softmax function, Adam, and batch-normalization are quite common and popular algorithms for deep neural network due to their superior performance compared to alternatives. However, efficient secure protocols for these have been elusive due to intractability of computing the elementary functions the depend on. The softmax function requires exponentiation and inversion, as shown in Eq. (8), and Adam and batch-normalization require the inverse of square roots, as shown in Eq. (9) and (7). Therefore, the softmax function has often been approximated by a different function [43], which always reduces accuracy, sometimes significantly [25], and only SGD, an elemental optimization method is used. Although FALCON realized the secure batch-normalization [57], it is not perfectly secure because it leaks the magnitude of $b = \sigma^2 + \epsilon$, i.e., α such that $2^\alpha \leq b < 2^{\alpha+1}$, to compute $\frac{1}{\sqrt{b}}$.

However, the efficient secure protocols for the elementary functions including exponentiation, division, inversion, and the inverse of square root presented in Section IV allow us to implement secure deep neural network using softmax, Adam, and batch-normalization, as opposed to resorting to approximations and less efficient learning algorithms.

We further prepare building blocks other than the softmax, Adam, and batch-normalization as follows. A popular process in layers is matrix multiplication. We apply [12] to compute the inner products with the same communication cost of a single multiplication. Since the ReLU′ function extracts the sign of the input, we can use the same approach as in Protocol 15. The ReLU function can be obtained by simply multiplying the input with the output of ReLU′. Secure max-pooling computes the maximum value and a flag (that is required in backpropagation) to indicate which is the maximum value by repeatedly applying the comparison protocol [28].

These building blocks are combined to form a secure deep learning system. More details can be found in Appendix J, which includes discussion about other ML related techniques.

VI. Experimental Evaluation

Environment. We implemented our protocols using $p = 2^{61} - 1$, and instantiated $\mathcal{F}_{\text{BC}}$, $\mathcal{F}_{\text{mod}}$, and $\mathcal{F}_{\text{QT}}$ with the bit-composition, modulus-conversion, and quotient transfer protocols from [28], respectively. We set the statistical security parameter for active-with-abort security to $\kappa = 8$.[3] All experiments are run in the

[3]While this parameter is relatively small compared to a somewhat more standard parameter like $\kappa = 40$ [3], an active adversary will have only a single chance to cheat for the implemented techniques [12], [27] and an honest party can detect it with probability over 99%.

execution environment shown in Table III, artificially limiting the network speed to 320Mbps and latency to 40ms when simulating a WAN setting.

A. Accuracy and Throughput

We measured the accuracy and throughput of our division protocol and elementary functions. Due to space limitation, the details of our experiments are deferred to Appendix K. In the following, we highlight our key findings.

- Output of our division protocol is close to real-valued division, with an L1-norm error of $0.335/2^t$ for input with offset t.
- All elementary functions have at least 23-bit accuracy.
- The throughput for the elementary functions are an order of magnitude faster than Sharemind [48] when processing 1M records.[4]

B. Secure Training of DNNs

We measured the performance of training the DNN architectures highlighted below. The parameters for Adam in all our experiments are $\beta_1 = 0.09, \beta_2 = 0.999, \eta = 2^{-10} (\eta' = 10), \epsilon = 0$, which are the recommended parameters in [29] (except ϵ).

Network Architectures We consider three networks in our experiments: (1) 3DNN, a simple 3-layer fully-connected network introduced in SecureML [43] and used as a benchmark for privacy preserving ML, (2) AlexNet, the famous winner of the 2012 ImageNet ILSVRC-2012 competition [33] and a network with more than 60 million parameters, and (3) VGG16, the runner-up of the ILSVRC-2014 competition [54] and a network with more than 138 million parameters. While the first network is typically used as a performance benchmark for privacy preserving ML, measurements with the latter two networks provide insight into the performance when using larger more realistic networks.

Datasets We use two datasets for our experiments: (1) MNIST [36], a collection of 28 x 28 pixel images of handwritten digits typically used for benchmarks, and (2) CIFAR-10 [32], a collection of colored 32 x 32 pixel images picturing dogs, cats, etc. We used MNIST in combination with 3DNN for benchmarking, and CIFAR-10 in combination with the larger networks AlexNet and VGG16.

Comparison For experimental comparison with related work, we will focus on the state-of-the-art three-party protocol, FALCON [57]. We note that the two-party protocols, SecureML [43] and Quotient [1], are outperformed by any of the three-party protocols by almost an order of magnitude in terms of running time, and among the three-party protocols, FALCON improves upon ABY3, which again is an improvement upon SecureNN. Furthermore, FALCON is the only other related work considering larger networks, AlexNet and VGG16.

Concretely, for all experiments, we ran the code from [57] in the same experimental environment and measured the execution time. We note that FALCON provides only 13-bit accuracy and sacrifices perfect security for performance, whereas our protocols provide 23-bit accuracy and perfect security. Furthermore, the code from [57] implements the online phase only, and hence, the measurements do not include the corresponding offline phase, which would make a significant contribution to the total running time. Lastly, the code does not update the parameters of the model, which makes the accuracy of the obtained model unclear. We emphasize that the measurements for our protocols are for the *total* running time and a fully trained model. Despite this, we treat the obtained measurements as comparable to ours. This is in favor of FALCON.

[4]Note that [48] seems like the most relevant comparison; ABY3 does not implement similar elementary functions, but replaces these with MPC-friendly functions.

TABLE IV: Comparison of training time of 3DNN over the MNIST dataset.

	Security/NW	Methods	Epochs	Time [s]	Accuracy [%]
FALCON	Passive/LAN	SGD	15	780	-
Ours		Adam	1	117	95.64
FALCON	Active/LAN	SGD	15	2,355	-
Ours		Adam	1	570	95.61
FALCON	Passive/WAN	SGD	15	16,110	-
Ours		Adam	1	4,537	95.64
FALCON	Active/WAN	SGD	15	37,185	-
Ours		Adam	1	11,516	95.61

Results for 3DNN We measured the running time and accuracy for training 3DNN on the MNIST dataset for passive and active security, in the LAN and WAN settings. The results for our protocols and FALCON can be seen in Table IV. Compared to FALCON, ours is between 3.2 to 6.7 times faster, depending on the setting. We again highlight that these results are achieved despite the advantages provided to FALCON in this comparison (measuring online time only, 13-bit vs. 23-bit accuracy, and imperfect security). For reference, we note that ours is only a factor of less than 6 slower than training *in the clear* using MLPClassifier from [46] (17.7 seconds, 95.54 % accuracy) on a single machine.

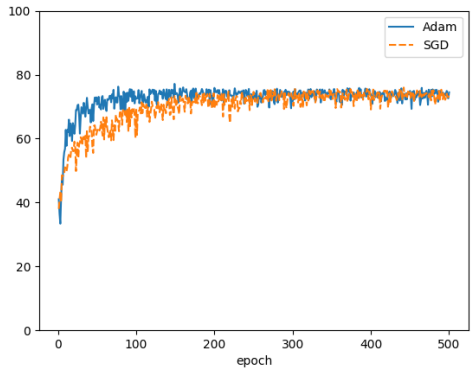
Results for AlexNet and VGG16 In the original paper of FALCON [57], the total running time for training on AlexNet and VGG16 was estimated through extrapolation since these networks require a significant amount of computation for training, even in the clear. We follow this method to estimate the running time of ours and FALCON (re-evaluated in our environment) in the same way.

In Table V, we show the measured running time to complete a single epoch for AlexNet and VGG16 using the CIFAR-10 dataset, both for passive and active security, as well as in the LAN and WAN settings. The table include measurements for both FALCON and our protocols. Note, however, that the time to complete a single epoch is not indicative of the overall performance difference between FALCON and our framework, as the underlying optimization methods are different, and require a different number of epochs to train a network achieving a certain prediction accuracy.

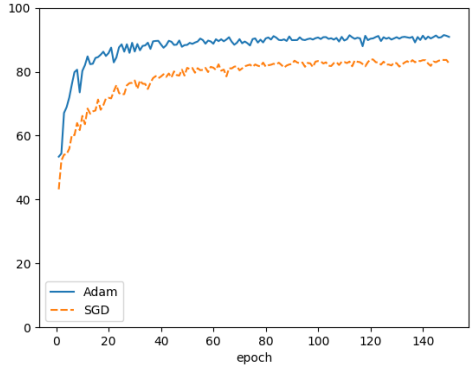
To determine the number of epochs needed for Adam (implemented in our framework) and SGD (implemented in FALCON), we ran Adam and SGD for AlexNet and VGG16 on CIFAR-10 in the clear, and measured the achieved accuracy.

TABLE V: Measured running time per epoch for training AlexNet and VGG16 on CIFAR-10.

	Security	Setting	AlexNet [s]	VGG16 [s]
FALCON	Passive	LAN	10,892	523,127
Ours	Passive	LAN	3,139	43,150
FALCON	Active	LAN	41,537	2,051,751
Ours	Active	LAN	15,021	161,481
FALCON	Passive	WAN	23,489	575,699
Ours	Passive	WAN	49,833	347,928
FALCON	Active	WAN	75,838	2,240,515
Ours	Active	WAN	159,781	1,293,226



(a) AlexNet



(b) VGG16

Fig. 1: Accuracy of AlexNet and VGG16 trained with Adam and SGD on CIFAR-10.

The results are illustrated in Figure 1. For AlexNet, we see that accuracy converges towards $75\% \sim 78\%$, with Adam achieving a maximum of 77.15% and SGD a maximum of 75.98% in our test. We note that Adam achieves an accuracy exceeding 70% after 25 epochs, whereas SGD requires 107 epochs. For VGG16, we see that Adam significantly outperforms SGD, and after relatively few epochs, achieves an accuracy not obtained by SGD, even after 150 epochs. We note that achieving an accuracy exceeding 75% requires 6 and 23 epochs for Adam and SGD, respectively, whereas an accuracy of 80% requires 8 and 45 epochs, respectively.

Based on the observations above, we estimate the running time of achieving an accuracy of 70% for AlexNet and 75% for VGG16, for active and passive security in the LAN and WAN setting. The result is shown in Table VI. We see that in the LAN setting, the total running time of our framework outperforms the online phase of FALCON with a factor of about $12 \sim 14$ for AlexNet and $46 \sim 48$ for VGG16, whereas in the WAN setting, the factors are about 2 and 6, respectively.

TABLE VI: Estimated running time for training of AlexNet (70% accuracy) and VGG16 (75% accuracy) on CIFAR-10.

	Security	Setting	AlexNet [h]	VGG16 [h]
FALCON	Passive	LAN	324	3,342
Ours	Passive	LAN	22	72
FALCON	Active	LAN	1,235	13,108
Ours	Active	LAN	104	269
FALCON	Passive	WAN	698	3,678
Ours	Passive	WAN	346	580
FALCON	Active	WAN	2,254	14,314
Ours	Active	WAN	1,110	2,155

The above comparison illustrates the advantage of the approach taken in our framework; by constructing efficient (and highly accurate) protocols that allow advanced ML algorithms such as Adam to be evaluated, despite these containing "MPC-unfriendly functions", we gain a significant advantage in terms of overall performance compared to previous works like FALCON, that attempt to achieve efficiency by simplifying the underlying ML algorithms, and optimizing the evaluation of these. As shown, the advantage when considering larger more realistic networks can in some cases be significantly more pronounced than suggested by the evaluation results on benchmark networks such as 3DNN, which is illustrated by the obtained 46 times faster evaluation of VGG16 in the LAN setting. We again highlight that this is obtained despite the advantages offered to FALCON in the comparison.

Note on Trident Finally, for completeness, we note that Trident [47] improves upon the online phase of ABY3 by increasing the number of servers to four and pushing more of the computation to the offline phase. Note that being a four-party protocol (tolerating a single corruption), Trident obviously increases cost in terms of the required number of servers, but also weakens security compared to the above mentioned three-party protocols, and is hence not directly comparable to our framework. Nevertheless, from the measurements which are provided in [47], we estimate that, for active security, the *online* phase of Trident is somewhat faster in the LAN setting and somewhat worse in the WAN setting compared to the *total* running time for our protocols when considering the simple 3DNN network.[5] However, including the offline phase, which is significant for Trident, will add considerably to the running time[6]. This strongly indicates that, despite being a four-party protocol, Trident offers worse overall performance

[5]From the measurements reported in [47], we estimate that the online phase of Trident in their environment would require 306s and 30264s to train 3DNN on the MNIST dataset in the LAN and WAN setting, respectively. The corresponding total time for our protocols are 570s and 11516s, respectively.

[6]Note that the offline phase in Trident is slower than the semi-honest ABY3 implementation (see [47][Appendix E.B]), which is already heavy. Furthermore note that the offline communication cost in Trident is the same or larger than in the online phase for all the 12 protocols in [47], except bit extraction.

than our protocols, in particular in the WAN setting, while at the same time relying on simplifications of the underlying ML learning algorithms. Additionally, Trident does not implement batch normalization required for AlexNet, and does not report any measurements for larger networks, for which we expect our framework to have a greater advantage. Since the source code is furthermore not available, such measurements are not easily obtainable.

VII. Conclusion

In this paper, we proposed a framework that enables efficient and secure evaluation of ML algorithms via three-party protocols for MPC-unfriendly computations. We first proposed a new division protocol, which enables efficient and accurate fixed-point arithmetic computation, and based on this, efficient protocols for machine learning, such as inversion, square root extraction, and exponential function evaluation. These protocols enable us to efficiently compute modern ML algorithms such as Adam and the softmax function as is. As a result, we obtain secure DNN training that outperforms state-of-the-art three-party systems in all tested settings, with the most pronounced advantage for large networks in the LAN setting.

References

[1] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-party secure neural network training and prediction," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 1231–1247.

[2] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the SPDZ compiler for other protocols," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 880–895.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 843–862. [Online]. Available: https://doi.org/10.1109/SP.2017.15

[4] 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 CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 805–817.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *20th ACM STOC*. ACM Press, May 1988, pp. 1–10.

[6] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery, "Efficient simd arithmetic modulo a mersenne number," in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 213–221.

[7] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *CRYPTO 2018, Part III*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10993. Springer, Heidelberg, Aug. 2018, pp. 483–512.

[8] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "Flash: Fast and robust framework for privacy-preserving machine learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 459 – 480, 2020. [Online]. Available: https://content.sciendo.com/view/journals/popets/2020/2/article-p459.xml

[9] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning," Cryptology ePrint Archive, Report 2017/1109, 2017, https://eprint.iacr.org/2017/1109.

[10] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: high throughput 3pc over rings with application to secure prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, R. Sion and C. Papamanthou, Eds. ACM, 2019, pp. 81–92. [Online]. Available: https://doi.org/10.1145/3338466.3358922

[11] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/trident-efficient-4pc-framework-for-privacy-preserving-machine-learning/

[12] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," in *CRYPTO 2018, Part III*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10993. Springer, Heidelberg, Aug. 2018, pp. 34–64.

[13] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, "An efficient secure three-party sorting protocol with an honest majority," *IACR Cryptology ePrint Archive*, vol. 2019, p. 695, 2019. [Online]. Available: https://eprint.iacr.org/2019/695

[14] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, and B. Pinkas, "High-throughput secure AES computation," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018, Toronto, ON, Canada, October 19, 2018*, M. Brenner and K. Rohloff, Eds. ACM, 2018, pp. 13–24. [Online]. Available: https://doi.org/10.1145/3267973.3267977

[15] R. Cramer, I. Damgård, and Y. Ishai, "Share conversion, pseudorandom secret-sharing and applications to secure computation," in *TCC 2005*, 2005, pp. 342–362.

[16] A. Dalskov, D. Escudero, and M. Keller, "Fantastic four: Honest-majority four-party secure computation with malicious security," Cryptology ePrint Archive, Report 2020/1330, 2020, https://eprint.iacr.org/2020/1330.

[17] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *CRYPTO 2007*, 2007, pp. 572–590.

[18] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified VSS and fact-track multiparty computations with applications to threshold cryptography," in *PODC*, 1998, pp. 101–111.

[19] 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 *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 201–210. [Online]. Available: http://proceedings.mlr.press/v48/gilad-bachrach16.html

[20] O. Goldreich, *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[21] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *19th ACM STOC*, A. Aho, Ed. ACM Press, May 1987, pp. 218–229.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[23] M. Ito, A. Saito, and T. Nishizeki, "Secret sharing scheme realizing general access structure," in *Proceedings IEEE Globecom '87*. IEEE, 1987, pp. 99–102.

[24] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, Aug. 2018, pp. 1651–1669.

[25] M. Keller and K. Sun, "Effectiveness of mpc-friendly softmax replacement," 2020.

[26] keras, https://keras.io/.

[27] R. Kikuchi, N. Attrapadung, K. Hamada, D. Ikarashi, A. Ishida, T. Matsuda, Y. Sakai, and J. C. N. Schuldt, "Field extension in secret-

shared form and its applications to efficient secure computation," in *ACISP 19*, ser. LNCS, J. Jang-Jaccard and F. Guo, Eds., vol. 11547. Springer, Heidelberg, Jul. 2019, pp. 343–361.

[28] R. Kikuchi, D. Ikarashi, T. Matsuda, K. Hamada, and K. Chida, "Efficient bit-decomposition and modulus-conversion protocols with an honest majority," in *ACISP 18*, ser. LNCS, W. Susilo and G. Yang, Eds., vol. 10946. Springer, Heidelberg, Jul. 2018, pp. 64–82.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[30] H. Kitai, J. P. Cruz, N. Yanai, N. Nishida, T. Oba, Y. Unagami, T. Teruya, N. Attrapadung, T. Matsuda, and G. Hanaoka, "MOBIUS: model-oblivious binarized neural networks," *IEEE Access*, vol. 7, pp. 139 021–139 034, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2939410

[31] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: super-fast and robust privacy-preserving machine learning," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 592, 2020. [Online]. Available: https://eprint.iacr.org/2020/592

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

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017. [Online]. Available: http://doi.acm.org/10.1145/3065386

[34] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow: Secure TensorFlow inference," in *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 336–353.

[35] S. Laur, J. Willemson, and B. Zhang, "Round-efficient oblivious database manipulation," in *ISC*, 2011, pp. 262–277.

[36] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[37] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via MiniONN transformations," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 619–631.

[38] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rkgz2aEKDr

[39] Q. Lou, B. Feng, G. C. Fox, and L. Jiang, "Glyph: Fast and accurately training deep neural networks on encrypted data," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/685ac8cadc1be5ac98da9556bc1c8d9e-Abstract.html

[40] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg3g2R9FX

[41] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, Aug. 2020, pp. 2505–2522.

[42] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 35–52.

[43] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 19–38. [Online]. Available: https://doi.org/10.1109/SP.2017.12

[44] K. Nandakumar, N. K. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 40–48. [Online]. Available: http://openaccess.thecvf.com/content_CVPRW_2019/html/CV-COPS/Nandakumar_Towards_Deep_Neural_Network_Training_on_Encrypted_Data_CVPRW_2019_paper.html

[45] A. Patra and A. Suresh, "BLAZE: blazing fast privacy-preserving machine learning," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/blaze-blazing-fast-privacy-preserving-machine-learning/

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[47] R. Rachuri and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *NDSS 2020*. The Internet Society, Feb. 2020.

[48] J. Randmets, "Programming languages for secure multi-party computation application development," Ph.D. dissertation, University of Tartu, 2017.

[49] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based oblivious deep neural network inference," in *USENIX Security 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, Aug. 2019, pp. 1501–1518.

[50] 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 *ASIACCS 18*, J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, Eds. ACM Press, Apr. 2018, pp. 707–721.

[51] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: scalable provably-secure deep learning," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2018, pp. 2:1–2:6. [Online]. Available: https://doi.org/10.1145/3195970.3196023

[52] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[53] scikit learn, https://scikit-learn.org/stable/.

[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[55] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2670313

[56] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," *PoPETs*, vol. 2019, no. 3, pp. 26–49, Jul. 2019.

[57] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: Honest-majority maliciously secure framework for private deep learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 1, pp. 188 – 208, 01 Jan. 2021. [Online]. Available: https://content.sciendo.com/view/journals/popets/2021/1/article-p188.xml

[58] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *23rd FOCS*. IEEE Computer Society Press, Nov. 1982, pp. 160–164.

APPENDIX A
DEFERRED DETAILS ON SHARE CONVERSIONS

This section provides details for share conversions, deferred from Section II-C. Our protocols will utilize the following share conversions.

$[\![a]\!] \to \langle\!\langle a \rangle\!\rangle$*:* Let $[\![a]\!]_i = (a_i, a_{i+1})$ be P_i's share for $i = 1, 2, 3$. The conversion from $[\![a]\!]$ to $\langle\!\langle a \rangle\!\rangle$, which we denote $\langle\!\langle a \rangle\!\rangle \leftarrow \mathsf{ConvertToAdd}([\![a]\!])$, is a local operation: P_1 and P_2 set $\langle\!\langle a \rangle\!\rangle_1 := a_1$ and $\langle\!\langle a \rangle\!\rangle_2 := a_2 + a_3$, respectively.

$\langle\!\langle a \rangle\!\rangle \to [\![a]\!]$*:* The conversion from $\langle\!\langle a \rangle\!\rangle$ to $[\![a]\!]$, which we denote $[\![a]\!] \leftarrow \mathsf{ConvertToRep}(\langle\!\langle a \rangle\!\rangle)$, is achieved via a simple protocol from [28]: P_1 and P_2 secret-share their shares $\langle\!\langle a \rangle\!\rangle_1$ and $\langle\!\langle a \rangle\!\rangle_2$ using a $[\![\cdot]\!]$-sharing, and send $[\![\langle\!\langle a \rangle\!\rangle_1]\!]_i$ and $[\![\langle\!\langle a \rangle\!\rangle_1]\!]_i$, respectively, to party P_i. Each P_i adds the received shares as $[\![a]\!]_i = [\![\langle\!\langle a \rangle\!\rangle_1]\!]_i + [\![\langle\!\langle a \rangle\!\rangle_2]\!]_i$ locally. We can further optimize this algorithm by using a pseudo-random number generator [13], to achieve a communication cost of two field elements among three parties in a single round.

$[\![a]\!] \to [a_\ell], \ldots, [a_1]$*:* This conversion decomposes a shared secret a into shares of its bit representation $a_\ell, \ldots, a_1$, which is also known as bit-decomposition. Note that we only decompose the least significant ℓ bits of a. The ideal functionality for bit-decomposition, which we denote $\mathcal{F}_{\text{BDC}}$, is given in Functionality 9. A protocol that securely computes $\mathcal{F}_{\text{BDC}}$ in the presence of passive and active adversaries appear in [28].

FUNCTIONALITY 9 ($\mathcal{F}_{BDC}$ – Bit decomposition):
Upon receiving $[\![a]\!]$, $\mathcal{F}_{\text{BDC}}$ reconstructs a, generates shares $([a_1], \ldots, [a_\ell])$, where $a = \sum_{i=1}^{\ell} 2^{i-1} a_i$ and sends $([a_1]_i, \ldots, [a_\ell]_i)$ to P_i.

$[a_\ell], \ldots, [a_1] \to [\![a]\!]$*:* This conversion constructs shares of a secret a from shares of its bit representation $a_\ell, \ldots, a_1$, which is also known as bit-composition. The ideal functionality for bit-composition, which we denote $\mathcal{F}_{\text{BC}}$, is given in Functionality 10. In Appendix I, we present an efficient bit-composition protocol based on quotient transfer presented below. This is a modified version for modulus p while the known bit composition protocol [2] works in only modulus 2^n.

FUNCTIONALITY 10 ($\mathcal{F}_{BC}$ – Bit composition):
Upon receiving $[a_1], \ldots, [a_\ell]$, $\mathcal{F}_{\text{BDC}}$ reconstructs $a_1, \ldots, a_\ell$, computes $a := \sum_{i=1}^{\ell} a_i \cdot 2^i$, generates shares $[\![a]\!]$, and sends $[\![a]\!]_i$ to P_i.

$[a] \to [\![a]\!]$*:* This conversion essentially changes the modulus of the underlying field of the shares while maintaining the secret i.e. shares $[a]_i \in \mathbb{Z}_2$ for a secret $a \in \{0, 1\}$ are converted to shares $[\![a]\!]_i \in \mathbb{Z}_p$. The ideal functionality of this modulus-conversion is given in Functionality 11. Protocols that securely compute $\mathcal{F}_{\text{mod}}$ in the presence of passive and active adversaries appear in [28].

FUNCTIONALITY 11 ($\mathcal{F}_{mod}$ – Modulus conversion):
Upon receiving $[a]$, $\mathcal{F}_{\text{mod}}$ reconstructs a, generates shares $[\![a]\!]$, and sends $[\![a]\!]_i$ to P_i.

APPENDIX B
QUOTIENT TRANSFER PROTOCOL

We describe the quotient transfer protocols for $[\![\cdot]\!]$, $\langle\!\langle \cdot \rangle\!\rangle$, and $[\cdot]$.

Protocol 10 Quotient Transfer for $\langle\!\langle \cdot \rangle\!\rangle$

Functionality: $\langle\!\langle q \rangle\!\rangle \leftarrow \mathsf{QT}(\langle\!\langle a \rangle\!\rangle)$
Input: $\langle\!\langle a \rangle\!\rangle$ where a is a multiple of 2.
Output: $\langle\!\langle q \rangle\!\rangle$ where $\langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2 = a + qp$
1: P_0 and P_1 secret-share LSBs of $\langle\!\langle a \rangle\!\rangle_1$ and $\langle\!\langle a \rangle\!\rangle_2$ in modulo 2, respectively. Let them be $[\langle\!\langle a \rangle\!\rangle_1^{(1)}]$ and $[\langle\!\langle a \rangle\!\rangle_2^{(1)}]$.
2: $[q] := [\langle\!\langle a \rangle\!\rangle_1^{(1)}] \oplus [\langle\!\langle a \rangle\!\rangle_2^{(1)}]$.
3: $[\![q]\!] \leftarrow \mathcal{F}_{\text{mod}}([q])$
4: $\langle\!\langle q \rangle\!\rangle \leftarrow \mathsf{ConvertToAdd}([\![q]\!])$
5: Output $\langle\!\langle q \rangle\!\rangle$

Protocol 11 Quotient Transfer for $[\![\cdot]\!]$

Functionality: $[\![q]\!] \leftarrow \mathsf{QT}([\![a]\!])$
Input: $[\![a]\!]$ where a is a multiple of 4.
Output: $[\![q]\!]$ where $a_1 + a_2 + a_3 = a + qp$
1: P_0 and P_1 *locally* generate shares of the second LSBs of a_1, a_2, and a_3 in modulo 2, respectively. Let them be $[a_1^{(1)}]$, $[a_1^{(2)}]$, $[a_2^{(1)}]$, $[a_2^{(2)}]$, $[a_3^{(1)}]$, and $[a_3^{(2)}]$.
2: $[q_1] := [a_1^{(1)}] \oplus [a_2^{(1)}] \oplus [a_3^{(1)}]$.
3: $[c] \leftarrow ([a_1^{(1)}] \oplus [a_3^{(1)}]) \cdot ([a_2^{(1)}] \oplus [a_3^{(1)}]) \oplus [a_3^{(1)}]$.
4: $[q_1] := [a_1^{(2)}] \oplus [a_2^{(2)}] \oplus [a_3^{(2)}] \oplus [c]$.
5: $[\![q_1]\!] \leftarrow \mathcal{F}_{\text{mod}}([q_1])$
6: $[\![q_2]\!] \leftarrow \mathcal{F}_{\text{mod}}([q_2])$
7: Output $[\![q]\!] := [\![q_1]\!] + 2[\![q_2]\!]$

Informally speaking, the main idea of this protocol is that if we use an odd prime and the secret's LSB is zero, the addition of the truncated shares' LSBs corresponds to q.

The quotient transfer protocol uses multiplication and modulus conversion protocols. In the protocol, we describe them as functionalities $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{mod}}$. Please see [28] for their instantiations.

In the presence of passive adversaries, we use $\langle\!\langle a \rangle\!\rangle$ as an input of the quotient transfer protocol. Because $\langle\!\langle a \rangle\!\rangle$ consists of two sub-shares, the quotient q is 0 or 1 and the (single) LSB must be 0.

In the presence of active adversaries, we use $[\![a]\!]$ as an input of the quotient transfer protocol. Because $[\![a]\!]$ consists of three sub-shares, the quotient q is 0, 1, or 2, and the second LSBs must be 0s to contain 2. The step 3 and the last term of step 4 come from the fact that the carry of a_1, a_2, a_3 is $(a_1 \oplus a_3)(a_2 \oplus a_3) \oplus a_3$. This protocol is secure against an active adversary with abort using general compiler such as [12]. Note, in the step 1, the "share of sub-shares" can be generated locally. For details, see Section 4.4 in [28].

We can also introduce the quotient transfer protocol for $[\cdot]$ [2]. This protocol computes the carry, which means whether or not there are at least two sub-shares are 1.

APPENDIX C
FUNCTIONALITIES OF DIVISION PROTOCOLS

We give the functionalities for division by a public value, $\mathcal{F}_{\text{div}}$ and $\mathcal{F}_{\text{div\_general}}$, corresponding to division in the specific and general case, respectively, in Functionality 12 and Functionality 13.

FUNCTIONALITY 12 ($\mathcal{F}_{div}$ – Division by a public value):
Upon receiving $\langle\!\langle a \rangle\!\rangle_1$ and $\langle\!\langle a \rangle\!\rangle_2$, $\mathcal{F}_{\text{div}}$ reconstructs $a \equiv \langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2$ and computes $r_a = a \mod d$, and $r_1 = \langle\!\langle a \rangle\!\rangle_1 \mod d$. Then, $\mathcal{F}_{\text{div}}$ sets b as follows:

- $b = a/d$ if $\Big((\langle\!\langle a \rangle\!\rangle_1 \leq a) \wedge (r_a < r_1)\Big) \vee \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge (r_a - 1 < r_1)\Big)$,
- $b = a/d + 1$ otherwise.

Then, $\mathcal{F}_{\text{div}}$ randomly picks $\langle\!\langle b \rangle\!\rangle_1 \leftarrow F_p$, sets $\langle\!\langle b \rangle\!\rangle_2 \equiv b - \langle\!\langle b \rangle\!\rangle_1$, and hands the parties P_1 and P_2 their shares $\langle\!\langle b \rangle\!\rangle_1$ and $\langle\!\langle b \rangle\!\rangle_2$, respectively.

FUNCTIONALITY 13 ($\mathcal{F}_{div\_general}$ – Division by a public value in general case):
Upon receiving $\langle\!\langle a \rangle\!\rangle_1$ and $\langle\!\langle a \rangle\!\rangle_2$, $\mathcal{F}_{\text{div}}$ reconstructs $a \equiv \langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2$ and computes $r_a = a \mod d$, $r_p = p \mod d$, and $r_1 = \langle\!\langle a \rangle\!\rangle_1 \mod d$. Then, $\mathcal{F}_{\text{div}}$ sets b as follows:

- $b = a/d$ if $\Big((\langle\!\langle a \rangle\!\rangle_1 \leq a) \wedge (r_a < r_1 \leq r_p)\Big) \vee \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge \big((r_a + r_p < r_1) \vee (r_a + r_p - d < r_1 \leq r_p)\big)\Big)$.
- $b = a/d + 1$ if $\Big((\langle\!\langle a \rangle\!\rangle_1 \leq a) \wedge \big((r_1 \leq r_a) \wedge (r_1 \leq r_p)\big) \vee \big((r_a < r_1) \wedge (r_p < r_1)\big)\Big) \vee \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge \big((r_p < r_1 \leq r_a + r_p) \vee (r_1 \leq r_a + r_p - d)\big)\Big)$.
- $b = a/d + 2$ if $(\langle\!\langle a \rangle\!\rangle_1 \leq a) \wedge (r_p < r_1 \leq r_a)$.

Then, $\mathcal{F}_{\text{div}}$ randomly picks $\langle\!\langle b \rangle\!\rangle_1 \leftarrow F_p$, sets $\langle\!\langle b \rangle\!\rangle_2 \equiv b - \langle\!\langle b \rangle\!\rangle_1$, and hands the parties P_1 and P_2 their shares $\langle\!\langle b \rangle\!\rangle_1$ and $\langle\!\langle b \rangle\!\rangle_2$, respectively.

Protocol 12 Quotient Transfer for $[\cdot]$

Functionality: $[q] \leftarrow \mathsf{QT}([a])$
Input: $[a]$ (no restriction of a)
Output: $[q]$ where $a_1 + a_2 + a_3 = a + 2q$
1: P_0 and P_1 *locally* generate shares of a_1, a_2, and a_3 in modulo 2, respectively. Let them be $[a_1]$, $[a_2]$, and $[a_3]$.
2: $[q] \leftarrow ([a_1] \oplus [a_3]) \cdot ([a_2] \oplus [a_3]) \oplus [a_3]$.
3: Output $[q]$

APPENDIX D
DIVISION PROTOCOL FOR SIGNED INTEGERS

We extend our division protocols to signed integers in Protocol 13.

Protocol 13 Secure Division by Public Value in $[\![\cdot]\!]$ with *Signed* Integers

Functionality: $[\![c]\!] \leftarrow \mathsf{Div}^{\mathsf{S}}_{(2,3)}([\![a]\!], d)$
Input: Share of dividend $[\![a]\!]$ and (public) divisor d, where $-2^{|p|-2} - r_\omega \leq a \leq 2^{|p|-2} - r_\omega - 1$
Output: $[\![c]\!]$, where $c \approx \frac{a}{d}$
Parameter: $\omega := \left\lceil \frac{2^{|p|-2}}{d} \right\rceil$ and r_ω such that $\omega d = 2^{|p|-2} + r_\omega$
1: $[\![b]\!] \leftarrow \mathsf{Div}_{(2,3)}([\![wd + a]\!], d)$
2: $[\![c]\!] \leftarrow [\![b]\!] - [\![w]\!]$
3: Output $[\![c]\!]$

APPENDIX E
PRECISE ANALYSIS OF OUR DIVISION PROTOCOL

A. Specific Case

First, we focus on an the important case (for our application) in which p is a Mersenne prime and d is a power of 2.

In that case, $r_p = d - 1$ and Eq. (3) is

$$\alpha_a - q + 1 + \frac{r_a + q(d-1) - r_1 - r_2}{d} + r_1/d = \alpha_a - q + 1 + \frac{r_a + q(d-1) - r_1 - r_2}{d} \quad (10)$$

since $r_1 < d$.

Next, we separate the cases of $q = 0$ and $q = 1$. If $q = 0$ (that means $\langle\!\langle a \rangle\!\rangle_1 \leq a$), Eq. (10) is

$$\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d}.$$

Since $\langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2 \equiv a$ and $q = 0$, the equation $(\alpha_1 d + r_1) + (\alpha_2 d + r_2) = \alpha_a d + r$ holds and converts to

$$\frac{r_a - r_1 - r_2}{d} = \alpha_1 + \alpha_2 - \alpha_a.$$

Since α_1, α_2, and α_a are integers, $r - r_1 - r_2$ must be a multiple of d. In addition, since r, r_1, and r_2 are less than d, $r_a - r_1 - r_2$ is either 0 or $-d$. Precisely,

$$\frac{r_a - r_1 - r_2}{d} = \begin{cases} -1 & \text{if } r_a < r_1 \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, in the case of $q = 0$, the output is as follows.

$$\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d} = \begin{cases} \alpha_a & \text{if } r_a < r_1 \\ \alpha_a + 1 & \text{otherwise} \end{cases}. \quad (11)$$

We then switch to the case of $q = 1$ (that means $a < \langle\!\langle a \rangle\!\rangle_1$). Eq. (10) is

$$\alpha_a + \frac{r_a + r_p - r_1 - r_2}{d}.$$

Since $\langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2 \equiv a$ and $q = 1$, $(\alpha_1 d + r_1) + (\alpha_2 d + r_2) = (\alpha_a d + r_a) + (\alpha_p d + r_p)$ and

$$\frac{r_a + r_p - r_1 - r_2}{d} = \alpha_1 + \alpha_2 - (\alpha_a + \alpha_p).$$

Since all the terms in the right equation are integers, $r_a + r_p - r_1 - r_2$ is a multiple of d. In addition, since r_a, r_1, and r_2 are less than d and $r_p = d - 1$, $r_a + r_p - r_1 - r_2$ can be either 0 or 1. Precisely,

$$\frac{r_a + r_p - r_1 - r_2}{d} = \begin{cases} 0 & \text{if } r_a - 1 < r_1 \\ 1 & \text{otherwise,} \end{cases}$$

and we have the following in the $q = 1$ case.

$$\alpha_a + \frac{r_a + r_p - r_1 - r_2}{d} = \begin{cases} \alpha_a & \text{if } r_a - 1 < r_1 \\ \alpha_a + 1 & \text{otherwise .} \end{cases} \quad (12)$$

Let b be the secret of our protocol's output and recall that $\alpha_a = a/d$. From Eq. (11) and 12, we conclude that

$$b = \begin{cases} a/d & \text{if } \begin{array}{l} \Big((\langle\!\langle a \rangle\!\rangle_1 \le a) \wedge (r_a < r_1)\Big) \\ \vee \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge (r_a - 1 < r_1)\Big) \end{array} \\ a/d + 1 & \text{otherwise} \end{cases} \quad (13)$$

B. General Case

On input an additive share $\langle\!\langle a \rangle\!\rangle$ and divisor d, we show that our division protocol outputs $\langle\!\langle a/d \rangle\!\rangle$, $\langle\!\langle a/d + 1 \rangle\!\rangle$, or $\langle\!\langle a/d + 2 \rangle\!\rangle$ for a general p and d.

We first consider the $q = 0$ case. In this case, Eq. (3) is

$$\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d.$$

For the term of $\frac{r_a - r_1 - r_2}{d}$, the same discussion in the specific case holds.

We then consider the last term $(r_1 - r_p + d - 1)/d$. If $r_1 - r_p \le 0$, $0 \le r_1 - r_p + d - 1 < d$ since $-(d-1) \le r_1 - r_p$. Otherwise, $d \le r_1 - r_p + d - 1 < 2d$ since $r_1 - r_p \le d - 1$. Therefore,

$$(r_1 - r_p + d - 1)/d = \begin{cases} 0 & \text{if } r_1 \le r_p \\ 1 & \text{otherwise} \end{cases} \quad (14)$$

Therefore, we have the following equation in the $q = 0$ case.

$$\begin{aligned} &\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d} + (r_1 + d - 1 - r_p)/d \\ &= \begin{cases} \alpha_a & \text{if } r_a < r_1 \le r_p \\ \alpha_a + 1 & \text{if } \begin{array}{l} \big((r_1 \le r_a) \wedge (r_1 \le r_p)\big) \\ \vee \big((r_a < r_1) \wedge (r_p < r_1)\big) \end{array} \\ \alpha_a + 2 & \text{if } r_p < r_1 \le r_a \end{cases}. \end{aligned} \quad (15)$$

Next, we consider the $q = 1$ case. In this case, Eq. (3) is

$$\alpha_a + \frac{r_a + r_p - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d.$$

The same discussion as in the specific case holds and $\frac{r_a + r_p - r_1 - r_2}{d}$ must be a multiple of d. However, r_p can be small in the general case, and it affects the possible values of this term; namely, $r_a + r_p - (r_1 + r_2)$ can be -1 in addition to 0 and 1. Precisely,

$$\frac{r_a + r_p - r_1 - r_2}{d} = \begin{cases} -1 & \text{if } r_a + r_p < r_1 \\ 0 & \text{if } r_a + r_p - d < r_1 \le r_a + r_p \\ 1 & \text{otherwise} \end{cases} \quad (16)$$

The term $(r_1 - r_p + d - 1)/d$ is the same as Eq. (14). Therefore, combining the conditions of Eq. (14) and 16,

$$\begin{aligned} &\alpha_a + \frac{r_a + r_p - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d \\ &= \begin{cases} \alpha_a & \text{if } (r_a + r_p < r_1) \vee (r_a + r_p - d < r_1 \le r_p) \\ \alpha_a + 1 & \text{if } (r_p < r_1 \le r + r_p) \vee (r_1 \le r_a + r_p - d) \end{cases}. \end{aligned} \quad (17)$$

Let b be the secret of our protocol's output and recall that $\alpha_a = a/d$. From Eq. (15) and (17), we conclude that $b = a/d$ if

$$\begin{array}{l} \Big((\langle\!\langle a \rangle\!\rangle_1 \le a) \wedge (r_a < r_1 \le r_p)\Big) \vee \\ \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge \big((r_a + r_p < r_1) \vee (r_a + r_p - d < r_1 \le r_p)\big)\Big), \end{array}$$

$b = a/d + 1$ if

$$\begin{array}{l} \Big((\langle\!\langle a \rangle\!\rangle_1 \le a) \wedge \big((r_1 \le r_a) \wedge (r_1 \le r_p)\big) \vee \big((r_a < r_1) \wedge (r_p < r_1)\big)\Big) \\ \vee \Big((a < \langle\!\langle a \rangle\!\rangle_1) \wedge \big((r_p < r_1 \le r_a + r_p) \vee (r_1 \le r_a + r_p - d)\big)\Big), \end{array}$$

and $b = a/d + 2$ otherwise.

Appendix F
Probability of Each Output for Random Shares

In this section, we specify how the output of our passively secure division protocol depends on d, a, p, $\langle\!\langle a \rangle\!\rangle_1$ (and their dependent variables α_a, r_a, α_p, r_p, $\langle\!\langle a \rangle\!\rangle_2$, r_1, and r_2). In this section, we assume that $\langle\!\langle a \rangle\!\rangle_1$ is uniformly random in mod p. This is the case for our application – multiply-then-truncate. An output of multiplication protocol is uniformly random share, and an input of the division protocol is always the output of a multiplication protocol.

A. Specific Case

As in the correctness discussion, we first focus on the case in which p is a Mersenne prime and d is a power of 2.

From Eq. (13), we count integers that satisfying $(\langle\!\langle a \rangle\!\rangle_1 \le a) \wedge (r_a < r_1)$ or $(a < \langle\!\langle a \rangle\!\rangle_1) \wedge (r_a - 1 < r_1)$. We observe that $(\langle\!\langle a \rangle\!\rangle_1 \le a) \wedge (r_a < r_1)$ is true if

$$\langle\!\langle a \rangle\!\rangle_1 \in \{md + n \mid 0 \le m \le \alpha_a - 1, r_a + 1 \le n \le d - 1\}.$$

The number of integers satisfying the above is $\alpha_a(d - r_a - 1)$. We then observe that $(a < \langle\!\langle a \rangle\!\rangle_1) \wedge (r_a - 1 < r_1)$ is true if

$$\begin{aligned} \langle\!\langle a \rangle\!\rangle_1 \in &\{md + n \mid \alpha_a \le m \le \alpha_p, r_a \le n \le d - 1\} \\ &\setminus \{\alpha_a d + r_a, \alpha_p d + d - 1\}. \end{aligned}$$

The number of integers satisfying the above is $(\alpha_p - \alpha_a + 1)(d - r_a) - 2$.

Therefore, the probability that the output is a/d is

$$\begin{aligned} &\frac{\alpha_a(d - r_a - 1) + (\alpha_p - \alpha_a + 1)(d - r_a) - 2}{p} \\ &= \frac{\alpha_p d + (d - 1) - \alpha_p r_a - \alpha_a + r_a - 1}{p} \\ &= \frac{p - r_a(\alpha_p - 1) - \alpha_a - 1}{p}, \end{aligned}$$

and the probability that the output is $a/d + 1$ is

$$1 - \frac{p - r_a(\alpha_p - 1) - \alpha_a - 1}{p} = \frac{r_a(\alpha_p - 1) + \alpha_a + 1}{p}$$

B. General Case

We show the probability of each output for general p and d, including when d is not a power of 2. The probability depends on two relations: magnitude relations between r and r_p, and d and $r_a + r_p$.

Case 1: $r_p < r_a$ and $r_a + r_p < d$

First, we consider the $q = 0$ case, *i.e.,* $\langle\langle a \rangle\rangle_1 \leq a$. If $r_p < r_a$, Eq. (15) equals to

$$\alpha_a + 1 + \frac{r_a - (r_1 + r_2)}{d} + (r_1 + d - 1 - r_p)/d = \begin{cases} \alpha_a + 1 & \text{if } (r_1 \leq r_p) \vee (r_a < r_1) \\ \alpha_a + 2 & \text{if } r_p < r_1 \leq r_a \end{cases} \quad (18)$$

Recall that $q = 0$ means $\langle\langle a \rangle\rangle_1 \leq a$ and $r_1 = \langle\langle a \rangle\rangle_1 \mod d$. Therefore,

$$\Pr[(q = 0) \wedge ((r_1 \leq r_p) \vee (r < r_1))] = \frac{(d - r_a + r_p)\alpha_a + r_p + 1}{p}$$

and

$$\Pr[(q = 0) \wedge (r_p < r_1 \leq r_a)] = \frac{(r_a - r_p)(\alpha_a + 1)}{p}$$

Next, we consider the $q = 1$ case, *i.e.,* $a < \langle\langle a \rangle\rangle_1$. If $r_a + r_p < d$, Eq. (17) equals to

$$\alpha_a + \frac{r_a + r_p - (r_1 + r_2)}{d} + (r_1 - r_p + d - 1)/d = \begin{cases} \alpha_a & \text{if } (r_a + r_p < r_1) \vee (r_1 \leq r_p) \\ \alpha_a + 1 & \text{if } (r_p < r_1 \leq r_a + r_p). \end{cases} \quad (19)$$

Therefore, if $r_p < r$,

$$\Pr[(q = 1) \wedge ((r_a + r_p < r_1) \vee (r_1 \leq r_p)] = \frac{(\alpha_p - \alpha_a)(d - r) - 1}{p}$$

and

$$\Pr[(q = 1) \wedge (r_p < r_1 \leq r_a + r_p)] = \frac{(\alpha_p - \alpha_a - 1)r_a + r_p}{p}$$

In summary, if $r_p < r_a$ and $r_a + r_p < d$,

$$\Pr[\text{output is } \alpha_a] = \Pr[(q = 1) \wedge (r_a + r_p < r_1) \vee (r_1 \leq r_p)] = \frac{(\alpha_p - \alpha_a)(d - r_a) - 1}{p}$$

and

$$\Pr[\text{output is } \alpha_a + 1] = \Pr\left[\begin{array}{l}((q = 0) \wedge ((r_1 \leq r_p) \vee (r_a < r_1))) \\ \vee ((q = 1) \wedge (r_p < r_1 \leq r_a + r_p))\end{array}\right] = \frac{(d - r_a + r_p)\alpha_a + r_p + 1}{p} + \frac{(\alpha_p - \alpha_a - 1)r_a + r_p}{p} = \frac{(d - 2r_a + r_p)\alpha_a + 2r_p + (\alpha_p - 1)r_a + 1}{p}$$

and

$$\Pr[\text{output is } \alpha_a + 2] = \Pr[(q = 0) \wedge (r_p < r_1 \leq r)] = \frac{(r_a - r_p)(\alpha_a + 1)}{p}$$

Case 2: $r_p < r_a$ and $d \leq r_a + r_p$

The $q = 0$ case is the same as the case 1.

If $d \leq r_a + r_p$, Eq. (17) equals to

$$\alpha_a + \frac{r_a + r_p - (r_1 + r_2)}{d} + (r_1 - r_p + d - 1)/d = \begin{cases} \alpha_a & \text{if } (r_a + r_p - d < r_1 \leq r_p) \\ \alpha_a + 1 & \text{if } (r_p < r_1) \vee (r_1 \leq r_a + r_p - d). \end{cases}$$

If $r_p < r_a$,

$$\Pr[(q = 1) \wedge (r_a + r_p - d < r_1 \leq r_p)] = \frac{(\alpha_p - \alpha_a)(d - r_a) - 1}{p}$$

and

$$\Pr[(q = 1) \wedge ((r_p < r_1) \vee (r_1 \leq r_a + r_p - d))] = \frac{(\alpha_p - \alpha_a - 1)r_a + r_p}{p}$$

In summary, if $r_p < r_a$ and $d \leq r_a + r_p$,

$$\Pr[\text{output is } \alpha_a] = \Pr[(q = 1) \wedge (r_a + r_p - d < r_1 \leq r_p)] = \frac{(\alpha_p - \alpha_a)(d - r_a) - 1}{p}$$

and

$$\Pr[\text{output is } \alpha_a + 1] = \Pr\left[\begin{array}{l}((q = 0) \wedge ((r_1 \leq r_p) \vee (r < r_1))) \\ \vee ((q = 1) \wedge ((r_p < r_1) \vee (r_1 \leq r_a + r_p - d)))\end{array}\right] = \frac{(d - (r_a - r_p))\alpha_a + r_p + 1}{p} + \frac{(\alpha_p - \alpha_a - 1)r_a + r_p}{p} = \frac{(d - 2r_a + r_p)\alpha_a + 2r_p + (\alpha_p - 1)r_a + 1}{p}$$

and

$$\Pr[\text{output is } \alpha_a + 2] = \Pr[(q = 0) \wedge (r_p < r_1 \leq r_a)] = \frac{(r_a - r_p)(\alpha_a + 1)}{p}$$

The above shows the probability of cases 1 and 2 are the same.

Case 3: $r_a \leq r_p$ and $d \leq r_a + r_p$

If $r_a \leq r_p$, Equation 15 equals to

$$\alpha_a + 1 + \frac{r_a - (r_1 + r_2)}{d} + (r_1 + d - 1 - r_p)/d = \begin{cases} \alpha_a & \text{if } r_a < r_1 \leq r_p \\ \alpha_a + 1 & \text{if } (r_1 \leq r_a) \vee (r_p < r_1) \end{cases}$$

Similar to the previous case, we have

$$\Pr[(q = 0) \wedge (r_a < r_1 \leq r_p)] = \frac{(r_p - r_a)\alpha_a}{p}$$

and

$$\Pr[(q = 0) \wedge ((r_1 \leq r_a) \vee (r_p < r_1))] = \frac{(d - r_p + r_a)\alpha_a + r_a + 1}{p}$$

The $q = 1$ case, if $r_a \leq r_p$,

$$\Pr[(q = 1) \wedge ((r_a + r_p < r_1) \vee (r_1 \leq r_p)] = \frac{(\alpha_p - \alpha_a)(d - r_a) - 1 + r_p - r_a}{p}$$

and

$$\Pr[(q=1) \wedge (r_p < r_1 \leq r_a + r_p)] = \frac{(\alpha_p - \alpha_a) r_a}{p}$$

In summary, if $r_a \leq r_p$ and $r_a + r_p < d$,

$$\begin{aligned}
&\Pr[\text{output is } \alpha_a] \\
&= \Pr\begin{bmatrix}((q=0) \wedge (r < r_1 \leq r_p)) \\ \wedge ((q=1) \wedge (r_a + r_p < r_1) \vee (r_1 \leq r_p))\end{bmatrix} \\
&= \frac{(r_p - r)\alpha_a}{p} + \frac{(\alpha_p - \alpha_a)(d-r) - 1 + r_p - r}{p} \\
&= \frac{(\alpha_p - \alpha_a)d + r_p\alpha_a - \alpha_p r_a - 1 + r_p - r}{p} \\
&= \frac{p - a + r_p\alpha_a - \alpha_p r_a - 1}{p}
\end{aligned}$$

and

$$\begin{aligned}
&\Pr[\text{output is } \alpha_a + 1] \\
&= \Pr\begin{bmatrix}((q=0) \wedge ((r_1 \leq r_a) \vee (r_p < r_1))) \\ \vee ((q=1) \wedge (r_p < r_1 \leq r_a + r_p))]\end{bmatrix} \\
&= \frac{(d - r_p + r_a)\alpha_a + r_a + 1}{p} + \frac{(\alpha_p - \alpha_a) r_a}{p} \\
&= \frac{(d - r_p)\alpha_a + (\alpha_p + 1) r_a + 1}{p} \\
&= \frac{a - r_p\alpha_a + r_a\alpha_p + 1}{p}
\end{aligned}$$

Case 4: $r_a \leq r_p$ *and* $d \leq r_a + r_p$

The $q = 0$ case is the same as the case 3. If $r_a \leq r_p$,

$$\begin{aligned}
&\Pr[(q=1) \wedge (r_a + r_p - d < r_1 \leq r_p)] \\
&= \frac{(\alpha_p - \alpha_a)(d - r_a) - 1 + r_p - r_a}{p}
\end{aligned}$$

and

$$\begin{aligned}
&\Pr[(q=1) \wedge ((r_p < r_1) \vee (r_1 \leq r_a + r_p - d))] \\
&= \frac{(\alpha_p - \alpha_a) r_a}{p}
\end{aligned}$$

In summary, the probability is the same as that in the case 3.

Appendix G
Division Protocol Secure against an Active Adversary with Abort

We show the division protocol secure against an active adversary with abort in Protocol 14. In addition to canceling $q\alpha_p$ out, we adjust the output to make the difference between $\frac{a}{d}$ and the output small by adding constants. Experimental analysis of the output of this protocol is provided in Sect. VI, which shows on average the relative error is about 0.5. Precise analysis of the output distribution will be provided in the full version.

Protocol 14 Actively Secure Division by Public Value

Functionality: $[\![c]\!] \leftarrow \mathsf{Div}^{\mathsf{Mal}}_{(2,3)}([\![a]\!], d)$
Input: $[\![a]\!]$ and d, where a and d are multiples of 4
Output: $[\![c]\!]$, where $c \approx \frac{a}{d}$

1: Let α_p and r_p be $p = \alpha_p d + r_p$, where $0 \leq r_p < d$.
2: $[\![q]\!] \leftarrow \mathcal{F}_{\text{QT}}([\![a]\!])$
3: $z := \begin{cases} 1 & \text{if } r_p \geq d/2 \\ 0 & \text{otherwise} \end{cases}$
4: Let a_i be a sub-share of $[\![a]\!]$, *i.e.,* $a_1 + a_2 + a_3 = a \mod p$
5: **for** $1 \leq j \leq 3$ **do**
6: P_j and P_{j+1} compute $b_j := \begin{cases} a_j + (d - r_p) + (d - r_p)/2 \text{ in } \mathbb{N} & \text{if } j = 0 \\ a_j & \text{otherwise} \end{cases}$
7: P_j and P_{j+1} set $b'_j := \begin{cases} b_j/d + 1 & \text{if } \frac{b_j}{d} - b_j/d \geq \frac{d}{2} \\ b_j/d & \text{otherwise} \end{cases}$
8: $[\![b']\!]_i := (b'_i, b'_{i+1})$ for $i = 1, 2, 3$
9: Output $[\![b']\!] - (\alpha_p + z)[\![q]\!] - 1$

Appendix H
Extension to Signed Values

Some of our protocols are only suitable for computation over unsigned values. To extend the domain of these to include signed values, we make use of a functionality that extracts the sign and the absolute value of a share. This allows us to do computation over the absolute value, and later adjust the result according to the sign. We denote by ExtSignAbs the function that extracts the sign and absolute value of the input, and functionality $\mathcal{F}_{\text{extsignabs}}$

Theorem 14: The protocol in Figure 15 securely implements $\mathcal{F}_{\text{extsignabs}}$ in the $(\mathcal{F}_{\text{BDC}}, \mathcal{F}_{\text{mod}})$-hybrid model in the presence of a passive adversaries.

Protocol 15 Extract Sign and Absolute for Signed Integer

Functionality: $([\![f]\!], [\![b]\!]) \leftarrow \mathsf{ExtSignAbs}([\![a]\!])$
Input: $[\![a]\!]$
Output: $([\![f]\!], [\![b]\!])$, where $f = 1$ if $0 \leq a$ and $f = -1$ otherwise, and the least ℓ bits of b is $|a|$.
Parameter: ℓ, where ℓ is the bit-length of a, *i.e.,* $-2^\ell < a < 2^\ell$.

1: $[\![a']\!] = 2^{\ell+1} + [\![a]\!]$
2: $([a_{\ell+1}], \ldots, [a_1]) \leftarrow \mathcal{F}_{\text{BDC}}([\![a]\!])$.
3: $[\![a_{\ell+1}]\!] \leftarrow \mathcal{F}_{\text{mod}}([a_{\ell+1}])$. ▷ $a_{\ell+1} = 1$ if a is positive and $a_{\ell+1} = 0$ otherwise
4: $[\![f]\!] := 2[\![a_{\ell+1}]\!] - 1$ ▷ $f = 1$ if a is positive and $f = -1$ otherwise
5: $[\![b]\!] := (1 - [\![a_{\ell+1}]\!])2^{\ell+1} + [\![f]\!] \cdot [\![a']\!]$
6: Output $([\![f]\!], [\![b]\!])$

Appendix I
Bit Composition Protocol

We will make use of functionalities for bit decomposition and composition of shared values, denoted $\mathcal{F}_{\text{BDC}}$ and $\mathcal{F}_{\text{BC}}$, respectively. The former was discussed in Sect. II. The known bit composition protocol [2] works in only mod 2^n, we, therefore, propose the bit composition protocol in mod p.

Protocol 16 Bit Composition

Functionality: $[\![\sum_{i<\ell} 2^i a_i]\!] \leftarrow \mathsf{BC}([a_1], \ldots, [a_\ell])$

Input: $[a_1], \ldots, [a_\ell]$

Output: $[\![\sum_{i=1}^{\ell} 2^i a_i]\!]$

Parameter: The bit-length of secret ℓ

1: Each P_i sets $[b_1]_i := [0]$ ▷ Set its sub-share as $(0, 0)$
2: $[a'_1] := [a_1]$
3: $[q'_1] \leftarrow \mathcal{F}_{\text{QT}}([a'_1])$
4: **for** $i = 2$ to ℓ **do**
5: $\quad [a'_i] := [a_i] - [q'_{i-1}] - [b_{i-1}]$
6: $\quad [b_i] := ((1 - [a_i]) + [b_{i-1}]) \cdot ([q_{i-1}] + [b_{i-1}]) + [b_{i-1}]$ ▷ $[b_i]$ is the borrow of i-th bits
7: $\quad [q'_i] \leftarrow \mathcal{F}_{\text{QT}}([a'_i])$
8: $[\![b_\ell]\!] \leftarrow \mathcal{F}_{\text{mod}}([b_\ell])$
9: $[\![q'_\ell]\!] \leftarrow \mathcal{F}_{\text{mod}}([q'_\ell])$
10: $[\![a']\!]_j \leftarrow \sum_{i \le \ell} 2^{i-1} [a'_i]_j \mod p$
11: Output $[\![a']\!] - 2^\ell([\![b_\ell]\!] + [\![q'_\ell]\!])$

In the bit-composition protocol, we want to obtain the composed value $[\![a]\!]$ on input of its binary representation $[a_1], ..., [a_\ell]$ by computing addition as $\sum_i 2^{i-1}[a_i]$. However, this computation does not work since each $[a_i]$ has the quotient q_i so $2^{i-1}(2q_i + a_i) = 2^i q_i + 2^{i-1} a_i$ is added at the i-th bit. The known bit-composition protocol in [2] cancel $2^i q_i$ out by computing q_i recursively for $i \le \ell - 1$, and they does not need to obtain q_i for $\ell < i$; their underlying secret sharing is on modulus 2^ℓ so $2^\ell q_i$ will be 0. However, this is not the case in modulus p so we have to yield another approach to cancel $2^\ell a_i$ out.

In our protocol, we resolve this problem by using the modulus conversion protocol. By converting shares from modulus 2 into p, $2q_i + a_i$ is changed into $pq_i + a_i$, which is a_i in modulus p. The protocol we use to implement the latter is shown in Figure 16. The round complexity of this protocol is $\ell + 1$ if we instantiate $\mathcal{F}_{\text{QT}}$ and $\mathcal{F}_{\text{mod}}$ by Protocol 12 and that in [28], where ℓ is the maximum bit-length of shared secret.

Theorem 15: The protocol in Figure 16 securely implements $\mathcal{F}_{\text{BC}}$ in the ($\mathcal{F}_{\text{QT}}$, $\mathcal{F}_{\text{mult}}$, $\mathcal{F}_{\text{mod}}$)-hybrid model in the presence of a passive adversary.

APPENDIX J
DETAILS OF SECURE DEEP NEURAL NETWORK

A. Notation

L denotes the layer number, and when the number of hidden layers is n, the input layer is $L = 0$ and the output layer is $L = n + 1$. The value $d_{L=j}$ denotes the number of neurons in layer j, and N_j^i denotes the i-th neuron in layer $L = j$. The strength of the coupling of neurons is described by parameters w_i. Learning is a process that (iteratively) updates the parameters to obtain the appropriate output.

In this section, the unit of processing is a matrix, so we use different notation. Let $A = (a_{i,j})$ denote a matrix, $A \pm B$ and $A \cdot B$ denote the matrix addition/subtraction and product, and $A \circ B$ denote the element-wise multiplication. When we apply an algorithm Func with each element $a_{i,j}$ in a matrix A, we describe it as $\mathsf{Func}(A)$, such as $\sqrt{A}$ and $\mathcal{F}_{\text{BDC}}(A)$.

B. Details of Adam

We introduce the main process used in Adam. The process is the same in each layer so we omit the layer index. A variable t indicates the iteration number of the learning process, *e.g.,* the value G_t denotes the gradient of the t-th iteration. In addition, $M, V, \hat{M}, \hat{V}$ are matrices of the same size as G, and M and V are initialized by 0. Here the superscript t, such as β^t, represents the t-th power of β. Adam proceeds as follows.

$$\begin{aligned} M_{t+1} &= \beta_1 M_t + (1 - \beta_1) G_t \\ V_{t+1} &= \beta_2 V_t + (1 - \beta_2) G_t \circ G_t \\ \hat{M}_{t+1} &= \frac{1}{1 - \beta_1^t} M_{t+1} \\ \hat{V}_{t+1} &= \frac{1}{1 - \beta_2^t} V_{t+1} \\ W_{t+1} &= W_t - \frac{\eta}{\sqrt{\hat{V}_{t+1}} + \epsilon} \circ \hat{M}_{t+1} \end{aligned} \tag{20}$$

where $\circ$ denotes the element-wise multiplication of matrices.

C. Secure Protocols for Neural Network

1) ReLU and ReLU′ functions: The ReLU′ function extracts the sign of the input as a bit value $b \in \{0, 1\}$. Hence, we can use the same approach as in Protocol 15 to implement this. The ReLU function can obtained by simply multiplying the input with the output of ReLU′. We show secure protocols for the ReLU′ and ReLU functions in Protocol 17 and 18.

Protocol 17 Secure ReLU′ Function

Functionality: $[\![Z]\!] \leftarrow \mathsf{ReLU'}([\![U]\!])$
Input: A matrix $[\![U]\!]$
Output: $[\![Z]\!]$, where each element $z_{i,j}$ in Z is 0 if $z_{i,j} \le 0$ and 1 otherwise
Parameter: ℓ, where each element in Z is between $2^{-\ell} + 1$ and $2^\ell - 1$.
1: $[\![U']\!] = 2^{\ell+1} + [\![U]\!]$ ▷ Add $2^{\ell+1}$ to each element
2: $([U^{(\ell+1)}], \ldots, [U^{(1)}]) \leftarrow \mathcal{F}_{\text{BDC}}([\![U]\!])$
3: $[\![Z]\!] \leftarrow \mathcal{F}_{\text{mod}}([U^{(\ell+1)}])$. ▷ Each element in $U^{(\ell+1)}$ is 1 if the corresponding element in U is positive and 0 otherwise
4: Output $[\![Z]\!]$

Protocol 18 Secure ReLU Function

Functionality: $[\![Y]\!] \leftarrow \mathsf{ReLU}([\![U]\!])$
Input: A matrix $[\![U]\!]$
Output: $[\![Y]\!]$, where $Y \leftarrow \mathbf{ReLU}(U)$.
Parameter: ℓ, where each element in U is between $2^{-\ell} + 1$ and $2^\ell - 1$.
1: $[\![Z]\!] = \mathsf{ReLU'}([\![U]\!])$
2: $[\![Y]\!] \leftarrow [\![Z]\!] \circ [\![U]\!]$

2) Softmax function: As shown in Eq. 8, the softmax function is computed via exponential and inversion functions, which we can implement via Protocol 9 and 4, respectively. We show a secure protocol for the softmax function in Protocol 19.

Protocol 19 Secure Softmax Function

Functionality: $[\![Y]\!] \leftarrow \mathsf{softmax}([\![U]\!])$
Input: A matrix $[\![U]\!] = ([\![u_{1,1}]\!], \ldots, [\![u_{m,n}]\!])$
Output: $[\![Y]\!]$, where $Y \leftarrow \mathbf{softmax}(U)$.

$[\![\overrightarrow{u_i}]\!] := ([\![u_{i,1}]\!], \ldots, [\![u_{i,n}]\!])$
for $i = 1$ to m (in parallel) **do**
  for $j = 1$ to n (in parallel) **do**
    Set $[\![\overrightarrow{u}_{i,j}]\!] = ([\![u_{i,j}]\!], \ldots, [\![u_{i,j}]\!])$ of length n ▷ all the elements are the same.
    $[\![\overrightarrow{b}_i]\!] \leftarrow [\![\overrightarrow{u}_i]\!] - [\![\overrightarrow{u}_{i,j}]\!]$
    $[\![\overrightarrow{c}_i]\!] \leftarrow \mathsf{Exponent}([\![\overrightarrow{b}_i]\!])$ ▷ $[\![\overrightarrow{c}_i]\!]$ is $([\![e^{u_{i,1}-u_{i,j}}]\!], \ldots, [\![e^{u_{i,n}-u_{i,j}}]\!])$
    $[\![\sum_{k=1}^{n} e^{u_{i,k}-u_{i,j}}]\!] := \sum_{k=1}^{n} [\![c_{i,k}]\!]$ ▷ Sum all the elements in C
    $[\![y_{i,j}]\!] \leftarrow \mathsf{Inv}([\![\sum_{k=1}^{n} e^{u_{i,k}-u_{i,j}}]\!])$
$[\![Y]\!] := ([\![y_{1,1}]\!], \ldots, [\![y_{m,n}]\!])$

3) Simultaneous offset management and right shift: The deep neural network computations are done using fixed-point arithmetic, and hence require truncation of the lower bits after each multiplication. At the same time, in the training process, we have to divide intermediate values with the batch size m, and additionally, results are scaled with the learning rate η, which is typically small *e.g.,* 0.001.

This opens up the possibility of an optimization in which the above operations, where appropriate, can be performed simultaneously using a single arithmetic right-shift. In particular, setting the batch size as a multiple of 2, such as $m = 2^7 = 128$, and using an approximate learning rate $\eta' := 2^{-10} \approx 0.001$, makes this optimization very simple to implement. Note that all of our protocols are designed to allow the input and output offsets to be chosen freely, and adding optimization such as the above, by adjusting the offsets, is trivial. In the following, we use the notation $\cdot_H$ to denote (matrix) multiplication with an additional right-shift H, and $\mathsf{InvSqrt}(\cdot, \nu)$ to denote computation of the inverse square root with an additional right-shift ν.

4) Secure Feedforward Deep Neural Networks with Adam: Let n be the number of hidden layers, $\widehat{\beta}_{1,t} = \frac{1}{1-\beta_1^t}$, and $\widehat{\beta}_{1,t} = \frac{1}{1-\beta_2^t}$. We show a protocol for secure feedforward deep neural networks with Adam in Protcool 20. In the protocol description, for brevity, we omit the division for signed and unsigned integers required by fixed-point multiplication. This should be done following the inner product and element-wise multiplication.

5) Extension to Convolutional Deep Neural Networks: In order to extend the feedforward neural network to the covolutional neural network, we need to implement a convolutional layer, batch normalization, and max-pooling. The convolutional layer can be computed as the usual fully connected layer, and batch-normalization can be computed trivially as long as the inverse of square root can be computed, since the rest is a linear operation. In the max-pooling, we need the maximum value in a vector and a flag indicating location of the maximum value to compute forward and backward propagation. We obtain those by repeatedly compare values in the vector.

The secure max-pooling is shown in Protocol 21. This protocol is defined as a recursive function. We define several

Protocol 20 Secure Feedforward Neural Network with Adam

Functionality: $[\![W^l]\!] \leftarrow \mathsf{FFNN\_Adam}([\![X]\!], [\![T]\!], [\![W^0]\!], \ldots, [\![W^n]\!], [\![M^0]\!], \ldots, [\![M^n]\!], [\![V^0]\!], \ldots, [\![V^n]\!])$
Input: Features of trained data $[\![X]\!]$, their label $[\![T]\!]$, initialized parameters $[\![W^l]\!]$, and vectors initialized by 0 $[\![M^l]\!]$ and $[\![V^l]\!]$ for $0 \le l \le n$
Output: Updated parameters $[\![W^l]\!]$ for $0 \le l \le n$
Parameter: $\eta', \beta_1, \beta_2, \hat{\beta}_{1,t}, \hat{\beta}_{2,t}$

—Forward propagation—
$[\![U^1]\!] \leftarrow [\![W^0]\!] \cdot [\![X]\!]$
$[\![Y^1]\!] \leftarrow \mathsf{ReLU}([\![U^1]\!])$
for $i = 1$ to $n-1$ **do**
  $[\![U^{i+1}]\!] \leftarrow [\![W^i]\!] \cdot [\![Y^i]\!]$
  $[\![Y^{i+1}]\!] \leftarrow \mathsf{ReLU}([\![U^{i+1}]\!])$
$[\![U^{n+1}]\!] \leftarrow [\![W^n]\!] \cdot [\![Y^n]\!]$
$[\![Y^{n+1}]\!] \leftarrow \mathsf{softmax}([\![U^{n+1}]\!])$
—Back propagation—
$[\![Z^{n+1}]\!] \leftarrow [\![Y^{n+1}]\!] - [\![T]\!]$
$[\![Z^n]\!] \leftarrow \mathsf{ReLU'}([\![U^n]\!]) \circ ([\![Z^{n+1}]\!] \cdot [\![W^n]\!])$
for $i = 1$ to $n-1$ **do**
  $[\![Z^{n-i}]\!] \leftarrow \mathsf{ReLU'}([\![U^{n-i}]\!]) \circ ([\![Z^{n-i+1}]\!] \cdot [\![W^{n-i}]\!])$
—Gradient evaluation—
$[\![G^0]\!] \leftarrow [\![Z^1]\!] \cdot_H [\![T]\!]$
for $i = 1$ to $n-1$ **do**
  $[\![G^i]\!] \leftarrow [\![Z^{i+1}]\!] \cdot_H [\![Y^i]\!]$
$[\![G^n]\!] \leftarrow [\![Z^{n+1}]\!] \cdot_H [\![Y^n]\!]$
—Parameter update by Adam—
for $i = 0$ to n **do**
  $[\![M^i]\!] \leftarrow \beta_1 [\![M^i]\!] + (1-\beta_1)[\![G^i]\!]$
  $[\![V^i]\!] \leftarrow \beta_2 [\![V^i]\!] + (1-\beta_2)[\![G^i]\!] \circ [\![G^i]\!]$
  $[\![\widehat{M^i}]\!] \leftarrow \widehat{\beta}_{1,t} [\![M^i]\!]$
  $[\![\widehat{V^i}]\!] \leftarrow \widehat{\beta}_{2,t} [\![V^i]\!]$
  $[\![\widehat{G^i}]\!] \leftarrow \mathsf{InvSqrt}([\![\widehat{V^i}]\!], \eta')$
  $[\![\widehat{G^i}]\!] \leftarrow [\![\widehat{G^i}]\!] \circ [\![\widehat{M^i}]\!]$
  $[\![W^i]\!] \leftarrow [\![W^i]\!] - [\![\widehat{G^i}]\!]$

functionalities and notations for this protocol: Let $\mathcal{F}_{\text{compare}}$ be the functionality on input $[\![\overrightarrow{a}]\!] = ([\![a_1]\!], \ldots, [\![a_n]\!])$ and $[\![\overrightarrow{b}]\!] = ([\![b_1]\!], \ldots, [\![b_n]\!])$ outputs $[\![\overrightarrow{c}]\!]$, where $c_i = 1$ if $a_i \ge b_i$ and 0 otherwise for $1 \le i \le n$. A parallel execution of a comparison protocol, e.g., [28], can realize $\mathcal{F}_{\text{compare}}$. Let $\mathcal{F}_{\text{CondAssignShare}}$ be the functionality that is similar to $\mathcal{F}_{\text{CondAssign}}$, but assigned values are not plaintexts but shares: on input of $([\![\overrightarrow{b}]\!], [\![\overrightarrow{a}]\!], [\![\overrightarrow{c}]\!])$ such that $c_i \in \{0, 1\}$, outputs $[\![\overrightarrow{d}]\!]$, where $d_i = a_i$ if $c_i = 1$ and $d_i = b_i$ otherwise for $1 \le i \le n$. This functionality can be realized in a similar way that computes $[\![c_i]\!] \cdot [\![a_i]\!] + (1 - [\![c_i]\!])[\![b_i]\!]$ using multiplication protocol in parallel. For two vectors $[\![\overrightarrow{e}]\!] = ([\![e_1]\!], \ldots, [\![e_n]\!])$ and $[\![\overrightarrow{f}]\!] = ([\![f_1]\!], \ldots, [\![f_m]\!])$, $[\![\overrightarrow{e}]\!] || [\![\overrightarrow{f}]\!]$ denote the concatenation of the vectors, i.e., $[\![\overrightarrow{e}]\!] || [\![\overrightarrow{f}]\!] = ([\![e_1]\!], \ldots, [\![e_n]\!], [\![f_1]\!], \ldots, [\![f_m]\!])$.

D. Possibilities of other ML-related algorithms

This section mentions some ML-related algorithms that can be implemented in secure computation, other than those mentioned in the text.

Shortcut [22] is a technique to add input not only to the next layer but also to a latter layer. This technique is easy to implement in secure computation by adding the input to the later layer.

Protocol 21 Secure max-pooling

Functionality: $([\![y]\!], [\![\overrightarrow{f}]\!]) \leftarrow \mathsf{MaxFlag}([\![\overrightarrow{x}]\!])$
Input: A vector of length n, $[\![\overrightarrow{x}]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$
Output: The maximum $[\![y]\!]$ and its original location $[\![\overrightarrow{f}]\!] = ([\![f_1]\!], \ldots, [\![f_n]\!])$, where $y = \max_{1 \le i \le n} x_i$ and $f_j = 1$ if $y = x_j$ and $f_j = 0$ otherwise for $1 \le j \le n$.[7]

1: **if** $n = 1$ **then return** $[\![x_1]\!], ([\![1]\!])$.
2: **else**
3: $m = n/2$
4: $[\![\overrightarrow{a}]\!] := ([\![x_1]\!], \ldots, [\![x_m]\!])$
5: $[\![\overrightarrow{b}]\!] := ([\![x_{m+1}]\!], \ldots, [\![x_n]\!])$
6: $[\![\overrightarrow{c}]\!] \leftarrow \mathcal{F}_{\text{compare}}([\![\overrightarrow{a}]\!], [\![\overrightarrow{b}]\!])$
7: $[\![\overrightarrow{d}]\!] \leftarrow \mathcal{F}_{\text{CondAssignShare}}([\![\overrightarrow{b}]\!], [\![\overrightarrow{a}]\!], [\![\overrightarrow{c}]\!])$
8: **if** $n \mod 2 = 0$ **then**
9: $([\![y]\!], [\![\overrightarrow{e}]\!]) \leftarrow \mathsf{MaxFlag}([\![\overrightarrow{d}]\!])$
10: $[\![\overrightarrow{z}]\!] \leftarrow \mathcal{F}_{\text{mult}}([\![\overrightarrow{e}]\!], [\![\overrightarrow{c}]\!])$
11: $[\![\overrightarrow{f}]\!] := [\![\overrightarrow{z}]\!] || ([\![\overrightarrow{e}]\!] - [\![\overrightarrow{z}]\!])$
12: **else**
13: $([\![y]\!], [\![\overrightarrow{e}]\!] || [\![r]\!]) \leftarrow \mathsf{MaxFlag}([\![\overrightarrow{d}]\!] || [\![x_n]\!])$
14: $[\![\overrightarrow{z}]\!] \leftarrow \mathcal{F}_{\text{mult}}([\![\overrightarrow{e}]\!], [\![\overrightarrow{c}]\!])$
15: $[\![\overrightarrow{f}]\!] := [\![\overrightarrow{z}]\!] || ([\![\overrightarrow{e}]\!] - [\![\overrightarrow{z}]\!]) || [\![r]\!]$

Dropout [55] is a method of dropping the information from random neurons to prevent overlearning. This method can also be implemented in secure computation. If it is public which neurons will be dropped, the parties generate a random number and set the weight of the neuron, corresponding to the random number, to 0. If we want to hide the location of a neuron to be dropped, the parties generate $[\![0]\!]$ or $[\![1]\!]$ for each neuron, shuffle them [35], and then multiply them by the weight.

Although we implemented Adam in this paper, other optimization methods, such as Adabound [40] and RAdam [38], consist of square root, reciprocal, and division. Therefore, it should be possible to implement these by using our seamless paradigm. This is a future work to implement them and compare their efficiency in the context of secure computation.

APPENDIX K
ACCURACY AND THROUGHPUT OF DIVISION AND ELEMENTARY FUNCTIONS

A. Accuracy

We compare the output of our secure division and other elementary protocols with the *real-valued* function *in the clear*.

Measuring Error in Division. We first give our experimental results on division and truncation, summarized in Table VII. We measured the error of the L1-norm as follows. We use inputs from 1 to n, where $n = 10,000$. The average-case and worst-case error refers to the value $\frac{1}{n}\sum_i \left|\frac{a_i - c_i}{c_i}\right|$ and $\max_i \left|\frac{a_i - c_i}{c_i}\right|$, respectively, where c_i is the correct output and a_i is the output of our protocol. From the table, we can observe that the output of our protocol is close to the real-valued division. For example, the L1 error of passively secure division protocol is 0.335 on average, meaning that the actual error is $\frac{0.335}{2^t}$ if the input offset is t.

Measuring Error in Elementary Protocols. We summarize the experimental results for error in elementary protocols in Table VIII. Again, we use inputs from 1 to n, where $n = 10,000$. We set the offset as $\ell = 10$. We set the divisor in the division with private divisor as 3, with offset being 0. Average-case and worst-case accuracy number refers to the value $-\log\left(\frac{1}{n}\sum_i \left|\frac{a_i - c_i}{c_i}\right|\right)$ and $-\log\left(\max_i \left|\frac{a_i - c_i}{c_i}\right|\right)$, respectively, where c_i is the correct output and a_i is the output of our protocol. This accuracy number x implies that the most significant x bits in the output of the protocols will be equal to those of a corresponding function in the clear. We prepare the correct output c_i by using functions implemented in the C Language with double precision.

TABLE VII: Accuracy of division and truncation

	Passive		Active	
	Average	Worst	Average	Worst
Truncation	0.3304	1.000	0.483	1.875
Division	0.335	1.059	0.495	2.000

TABLE VIII: Accuracy of elementary functions

	Passive		Active	
	Ave [bit]	Worst [bit]	Ave [bit]	Worst [bit]
Inversion	29.62	27.27	28.97	26.77
Division (private diviser)	29.61	27.2	28.99	26.59
Square root	29.33	27.02	28.88	26.64
Inverse of square root	29.34	27.05	28.86	26.55
Exponential	25.75	24.1	25.34	23.13

In the implementation, we use an internal precision that represents how many bits we used to represent intermediate values to obtain high accuracy of the output. The accuracy also depends on how many terms we compute of the Taylor of Newton series. In this experiment, we used 29-bit internal precision and computed by the 4-th term of the Taylor series for the inversion, 28-bit and the 6-th term for the inverse of square root, 25-bit and the 4-th term for exponential functions. The threshold for our hybrid table-lookup/series-expansion technique for exponential functions is set to $t = 4$.

The results show that our protocols achieve more than 23-bit accuracy, even in the worst case. Therefore, our protocols of elementary functions is as accurate as the Single-precision number, which also have 23-bit accuracy.

B. Throughput

We show the throughput of our protocol in Table IX. Note that for throughputs, higher is better. A secret of 29-bit length is used on this experiment. We listed the throughput results of state-of-the-art protocols from the latest version of Sharemind [48] as a reference.[8] In the settings where the number of records is large, our implementation is an order of magnitude faster than [48], while, in the setting with a small number of records, they are comparable.

[8]It is difficult to compare throughputs directly. They used a different machine and a similar but different modulus, and their throughput is for floating-point arithmetic. The last comes from the fact that the throughput of the fixed-point arithmetic is *slower* than that of the floating-point arithmetic in [48].

TABLE IX: Throughput of elementary functions [M op/s]

		Passive		Active
		1,000 [records]	1,000,000	1,000,000
Ours	Truncation	0.612	14.80	4.51
	Inversion	0.0219	1.179	0.236
	Square root	0.0147	0.608	0.136
	Exponential	0.0355	1.017	0.237
[48]	Truncation	0.833	1.82	-
	Inversion	0.0547	0.0567	-
	Square root	0.0559	0.0574	-
	Exponential	0.0350	0.0391	-

APPENDIX L
ADDITIONAL EXPERIMENTS WITH VARYING PARAMETERS.

In this section, we describe additional experiments, deferred from Section VI. We measured the running time when varying parameters such as the batch size, the number of neurons, the number of hidden layers, and the number of input records. These numbers are useful to estimate the running time for different networks and data.

Varying Batch Sizes. We show how the running time (for 1 epoch) depends on batch sizes when $n = 2, d_1 = d_2 = 128$ in Table X. Execution time for 1 epoch is almost in inverse proportion to batch size. Therefore, increasing the batch size helps to implement faster learning. This likely comes from the fact that the large batch size implies fewer parameter updates.

TABLE X: Execution Time for each Batch Size

Batch Size	64	128	256	512
Execution Time [s]	225	117	64	37

Varying the Number of Neurons. We show the running time (for 1 epoch) depends on the number of neurons when $n = 2, m = 128$ in Table XI. Even for a large number of neurons, *i.e.,* 256, the execution time is only 198 seconds.

TABLE XI: Execution Time for each # of Neurons

# of Neurons	32	64	128	256
Execution Time	68	85	117	304

Varying the Number of Hidden Layers. We show how the running time (for 1 epoch) depends on the number of hidden layers when $m = 128, d_1 = d_2 = 128$ in Table XII. For 4 hidden layers, the execution time is less than 3 minutes. This result roughly shows that we need approximately 15 seconds more to have another hidden layer. This helps us estimate execution time for deeper networks.

TABLE XII: Execution Time for each # of Hidden Layers

# of Hidden Layers	1	2	3	4
Execution Time	102	117	133	149

Large Data: 100 Features $\times$ 10 Million Data Samples. Table XIII shows the execution time per epoch using 100 attributes $\times$ 10 million data samples. The parameters $n = 2, d_0 = 100, d_1 = d_2 = 128, d_3 = 10$ are used.

TABLE XIII: Execution Time for 100 Attributes and 10 Million Data [s]

Batch Size	Our Protocol
512	$4,047$ [s] (1.12 [h])
1024	$2,724$ [s] (0.76 [h])