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

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 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 ~ 14 faster for AlexNet and 40 ~ 45 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 division 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, \(-\frac{r}{d}\), sneaks into the output when \(x + r \gg d\) 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 \(x_1 + x_2\) 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 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 \(\frac{r}{d}\)). Known efficient truncation protocols, e.g., [42], [43], use a table-lookup series-expansion to the respective parts. The details of how these techniques are used in our protocols differ depending on the size of the underlying ring/field, \(x\) becomes \(\frac{1}{\sqrt{p}}\). 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, \(-\frac{r}{d}\), sneaks into the output when \(x + r \gg d\) 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 \(x_1 + x_2\) 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 preprocessing 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 on 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 ∈ Z, we denote \( a/b \) ∈ R real-valued division, and by \( a/b \in \mathbb{Z} \) integer division that discards the remainder. In other words, \( a/b := \lfloor a/b \rfloor \).

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 \( \mathbb{Z}_2 \).
- \( \ell \)-bit unsigned and signed integers, \( \mathbb{Z}_\ell^{\alpha} = \{ a \in \mathbb{Z} \mid 0 \leq a < 2^{\ell-1} \} \) and \( \mathbb{Z}_\ell = \{ a \in \mathbb{Z} \mid -2^{\ell-1} \leq a \leq 2^{\ell-1} - 1 \} \), respectively, where the range of values for signed integers reflect that a single bit is used to indicate the sign.
- \( \ell \)-bit fixed-point unsigned and signed rational numbers \( \mathbb{Q}_\ell^{\alpha} = \{ b \in \mathbb{Q} \mid b = a/2^\ell, a \in \mathbb{Z}_\ell^{\alpha} \} \) and \( \mathbb{Q}_\ell = \{ b \in \mathbb{Q} \mid b = a/2^\ell, a \in \mathbb{Z}_\ell \} \), 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 \( p > 2^\ell \). This implies that signed integers are represented using ones’ compliment (i.e. a negative value \(-a \in \mathbb{Z}_p\) 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) \( \ell \)-bit integers \( 0 \leq a \leq 2^\ell - 1 \), which we denote \( \bar{\mathbb{Q}}_\ell^{\alpha} \), to represent the values \( \{ a/2^\ell \mid 0 \leq a \leq 2^\ell - 1 \} \), and will refer to \( \alpha \) as the offset for these. For a fixed-point value \( a \), we will use the notation \( a_\alpha \) to denote the integer representation with offset \( \alpha \) i.e. \( a_\alpha = a \cdot 2^\alpha \). The integers in \( \bar{\mathbb{Q}}_\ell^{\alpha} \) are represented as elements of \( \mathbb{F}_p \), and we denote the signed extension by \( \mathbb{Q}_\ell^{\alpha} \).

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

Finally note that since \( p = 2^\ell - 1 \) 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.

<table>
<thead>
<tr>
<th>System</th>
<th>Secure Capability</th>
<th>Supported ML Algorithms</th>
<th>Threat Model</th>
<th>Based Techniques</th>
<th>LAN/WAN</th>
<th>Evaluation Dataset</th>
<th>Network Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinNNN [37]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Chameleon [50]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>EzPC [9]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Gazette [24]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SecureML [41]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XONN [49]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Quotient [1]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Delphi [41]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>FHIE-based SGD [44]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Glyph [39]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>ABY3 [42]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SecureNN [56]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>CryptFlow [34]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>QuantizedNN [16]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>ASTRA [10]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>BLAZE [45]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Falcon [57]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>This work</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

"Basic" 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 Quotient, which uses also MotionSense, Thyroid, and more, besides MNIST). "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. \( P_1^3 p = a_3 2^3 + a_2 \mod p = a_0 + a_1 \mod p \) holds since \( 2^3 - 1 = 0 \mod p \). 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 \( i' \equiv i \mod 3 \) 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 parameterized by the statistical security parameter \( \kappa \), 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:

\([\cdot]\text{-sharing} : \) the 2-out-of-3 replicated sharing in \(\mathbb{Z}_p\),
\(\lbrack\cdot\rbrack\text{-sharing} : \) the 2-out-of-3 replicated sharing in \(\mathbb{Z}_2\),
\(\langle\langle\cdot\rangle\rangle\text{-sharing} : \) the simple additive sharing in \(\mathbb{Z}_p\).

\([\cdot]\text{-sharing}. \) 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 \([a]_i = (a_i, a_{i+1})\) as the \(P_i\)'s share for \(i = 1, 2, 3\). Denote \([a] = ([a]_1, [a]_2, [a]_3)\).

- Reconstruct: Given a pair of shares of \(a\), this protocol with passive adversary guarantees that all parties eventually obtain \(a\). With an active adversary, this functionality proceeds the same unless \([a]\) is not consistent, where all the honest parties will abort at the end of the execution.

- Local operations: Given shares \([a]\) and \([b]\) and a scalar \(\alpha \in \mathbb{Z}_p\), the parties can generate shares of \([a + b], [\alpha a], \) and \([\alpha + a]\) using only local operations. The notations \([a] + [b], [\alpha a], \) and \([\alpha + a]\) denote these local operations, respectively.

- Multiplications: Given shares \([a]\) and \([b]\), the parties can generate \([ab]\) by multiplying \([a]\) and \([b]\) via \(\mathcal{F}_{\text{mult}}\).

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

\(\lbrack\cdot\rbrack\text{-sharing}. \) This is exactly as \([\cdot]\text{-sharing} \) 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 \([a \land b]\) by multiplying \([a]\) and \([b]\) via \(\mathcal{F}_{\text{mult}}\).

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

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 \((a_1, \ldots, a_1)\) 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>
<thead>
<tr>
<th>Conversion</th>
<th>Functionality Name</th>
<th>Protocol</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>([a]) (\rightarrow) (\langle a\rangle)</td>
<td>ConvertToAdd</td>
<td>local operations</td>
<td>0</td>
</tr>
<tr>
<td>(\langle a\rangle) (\rightarrow) ([a])</td>
<td>ConvertToRep</td>
<td>one ([\cdot]\text{-sharing})</td>
<td>1</td>
</tr>
<tr>
<td>([a], \ldots, [a])</td>
<td>(\mathcal{F}_{\text{BC}}) – Bit decomposition</td>
<td>[28]</td>
<td>(\ell + 1)</td>
</tr>
<tr>
<td>(\langle a\rangle_1, \ldots, \langle a\rangle_1) (\rightarrow) ([a])</td>
<td>(\mathcal{F}_{\text{BC}}) – Bit composition</td>
<td>[28]</td>
<td>(\ell + 1)</td>
</tr>
<tr>
<td>([a]) (\rightarrow) ([a])</td>
<td>(\mathcal{F}_{\text{mod}}) – Modulus conversion</td>
<td>[28]</td>
<td>1</td>
</tr>
</tbody>
</table>

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

D. Quotient Transfer Protocol

Consider the reconstruction of shared secret, \(\langle a\rangle\) or \(\langle\langle a\rangle\rangle\), over \(\mathbb{N}\) as opposed to \(\mathbb{Z}_p\) for which \([\cdot]\) 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 \(\langle\langle\cdot\rangle\rangle\text{-sharing}\) for passive security and \([\cdot]\text{-sharing}\) 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 \(\langle a\rangle\) and \([a]\) 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 \([\cdot]\text{-sharings},\) 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 \(\langle a\rangle\), but we additionally use this functionality for \([a]\) and \([a]\). Note that for \(\langle a\rangle, q \in \{0, 1\}\) where \(\langle a\rangle_1 + \langle a\rangle_2 = a + qp\); for \([a]\), \(q \in \{0, 1, 2\}\) where sub-shares \(a_1 + a_2 + a_3 = a + qp\); and for \([a]\), \(q \in \{0, 1\}\) where sub-shares \(a_1 + a_2 + a_3 = a + 2q\).

**FUNCTIONALITY 1 (\(\mathcal{F}_{\text{QT}}\) – Quotient transfer):**

Upon receiving \(\langle a\rangle\), \(\mathcal{F}_{\text{QT}}\) sets \(q\) such that \(\langle a\rangle_1 + \langle a\rangle_2 = a + qp\) in \(\mathbb{N}\), generates shares \(\langle q\rangle\), and sends \(\langle q\rangle\) to \(P_1\).

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 \(2^\ell\), 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 $a = \alpha d + r_a$ 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 $\alpha_a$, or perhaps $\alpha_a \pm 1$.

The typical protocol proceeds as follows. The parties first prepare a shared correlated randomness $(\langle s' \rangle, \langle s \rangle)$, where $s' \in \mathbb{Z}_p$ and $s := s'/d$. (Note that $d$ is public and known a priori.)

The parties then compute $\langle a + s' \rangle$, reconstruct $(a + s')$, and set $\langle b \rangle = \langle a + s' \rangle/d$. Let $\langle s \rangle$, $\langle s' \rangle$ be sub-shares of input $s$ 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 $\langle a \rangle$, we locally convert $\langle a \rangle$ into $\langle a \rangle$ before division. Hence, in the following, we assume the input is $\langle a \rangle$ and a public divisor $d$.

First, let us analyze what happens when we simply divide each share by $d$. Let $\langle a \rangle_1 + \langle a \rangle_2 = qp + a \in \mathbb{N}$, where $q \in \{0, 1\}$. Here, suppose that $\langle a \rangle_j = \alpha_j d + r_j$, $a = \alpha_a d + r_a$, and $p = \alpha_p d + r_p$ for $j = 1, 2$ If each party divide its share $\langle a \rangle_j$ by $d$, the new share is $\alpha_j$, i.e., $\langle a \rangle_j/d = \alpha_j$. Then, the reconstruction of $(\alpha_1, \alpha_2)$ will be

$$\alpha_1 + \alpha_2 = \alpha_a + q \alpha_p + (r_a + qr_p - (r_1 + r_2))/d,$$

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

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 $\langle q \rangle$ 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 $\langle q \rangle$ 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 $\langle a \rangle$, where $a$ is a multiple of 2, and the second for $\langle a \rangle$ 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):

\[
(⟨⟨b⟩⟩_1 - (α_p + 1)⟨⟨q⟩⟩_1 + 1) + (⟨⟨b⟩⟩_2 - (α_p + 1)⟨⟨q⟩⟩_2) = ⟨⟨b⟩⟩_1 + ⟨⟨b⟩⟩_2 - (α_p + 1)(⟨⟨q⟩⟩_1 + ⟨⟨q⟩⟩_2) + 1
\]

\[
= (⟨⟨a⟩⟩_1 + d - 1 - r_p)/d + (⟨⟨α⟩⟩_2)/d - (α_p + 1)q + 1
\]

\[
= (α_1d + r_1 + d - 1 - r_p)/d + (α_2d + r_2)/d - qα_p - q + 1 = α_1 + α_2 - qα_p - q + 1 + (r_1 - r_p + d - 1)/d + r_2/d
\]

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

\[
α_α = q + 1 + \frac{r_α + qα_p - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d. \tag{3}
\]

The above computation shows the term \( qα_p \) being eliminated. The remaining terms are quite small since \( q \in \{0, 1\} \), each \( r_1 \) is smaller than \( d \), and \( \frac{r_α + qα_p - r_1 - r_2}{d} \) 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 F, 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.


**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 communication 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 \( [a] \)) and local computation in step 5 in Protocol 1 is performed over \( [\cdot] \)-sharings by computing ConvertToRep(\( ⟨⟨b⟩⟩ \) ) just before this step.

6) Comparison: We compare Protocol 2 with the one (denoted truncation in the original paper) of AYB3 [42], which has been used in subsequent literature, such as FALCON. Since the AYB3 protocol is proposed as truncation, we let \( d = 2^\ell \). The AYB3 protocol requires \( 6(2|p| - δ - 1) \) 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 \( 15|p| + 6δ - 6 \) 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 AYB3’s, and slightly worse than AYB3’s online cost. In addition, the AYB3’s protocol can contain a large error \( \frac{2}{q} \) 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 \( qα_p \) 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 \( [\cdot] \), where \( a \) is required to be a multiple of 4. Hence, our actively secure division protocol works directly on \( [\cdot] \) and essentially executes the following steps:

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

In addition, we further adjust the output using appropriate constants to minimize the difference between \( \frac{2}{q} \) and the protocol output. The concrete protocol appears in Appendix G.

Regarding security, as we do not convert to \( ⟨⟨a⟩⟩ \), 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 \( a ×_\ell b \), corresponds to (standard) multiplication of \( a \) and \( b \), followed by division by \( 2^\ell \). To ease notation in the protocols, we use \( [a] \cdot [b] \) to denote \( \mathcal{F}_\text{mult}([a],[b]) \), and \( [a] ×_\ell [b] \) to denote \( \mathcal{F}_\text{div}([a],[b],2^\ell) \).

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 \( \sum_{j=0}^{e_j} e^{x_j} = \sum_{j=0}^{e_j} \frac{1}{e^{x_j}} \) 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 \( \sum_{j=0}^{e_j} e^{x_j} \) 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 \(\alpha\) and \(\delta\) 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 \(\ell\) 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 \(\ell\) 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 as \(MSNZBFit\) (MSNZB denoting Most Significant Non-Zero Bit) and the corresponding functionality \(F_{msnzb} \). The protocol presented in Protocol 3 implements this functionality. Recall that \(F_{BC}\) and \(F_{BCD}\) are the functionalities of bit-(de)-composition.

**Theorem 3:** Protocol 3 securely implements \(F_{msnzb}\) in the \((F_{BCD}, F_{BC}, F_{mult})\)-hybrid model in the presence of a passive adversaries.

**Protocol 3 MSNZB Fitting**

**Functionality:** \(([b], [c]) \leftrightarrow MSNZBFit([a])\)

**Input:** \([a]\)

**Output:** \([b], [c]\), where \([b] = [a] \cdot [c], 2^\ell-1 \leq b \leq 2^\ell - 1\), and \(c = 2^\delta\) for some \(\ell \in \mathbb{N}\).

**Parameter:** \(\ell\)

1. \(([a_1], \ldots, [a_k]) \leftrightarrow F_{BCD}([a])\)
2. \([f_i] ::=[a_i]\)
3. for \(i = 0\) to \(\ell - 1\) do
4. \([f_i] := [f_{i+1}] \land [a_i] \triangleright f_i = 1\) for all \(i\) corresponding to MSNZB of \(a\) or smaller
5. \([x_i] ::=[a_i]\)
6. for \(i = \ell - 1\) to 1 do
7. \([x_i] := [f_i] \oplus [f_{i+1}] \triangleright x_i = 1\) only for \(i\) corresponding to MSNZB of \(a\)
8. \([c] \leftrightarrow F_{BC}([x_1], \ldots, [x_k])\) \triangleright Bit-compose \([x_i]\) in the reverse order to obtain \(c = 2^{\ell-1} \cdot \log_2 a\)
9. \([b] = [a] \cdot [c]\)
10. Output \([b]\) and \([c]\)

Using 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 \(0\), for \(x \in [0; \frac{1}{2})\):

\[
1 - x_1 = \sum_{i=0}^{\infty} x_1^i = 1 + x_1 + x_1^2 + \cdots .
\]

Continuing this Taylor series until the \(n\)-degree, yields the remainder term \(\frac{x_1^n}{1-x_1} \leq \frac{1}{2^n}\), which implies that the approximation has \(n\) bits accuracy.

Firstly, we use 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 \(\ell\) 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.

\[
\prod_{j=0}^{\infty} (1 + x_j^j) = (1 + x_1)(1 + x_1^2)(1 + x_1^4)\cdots .
\]

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 \([\frac{1}{b}]\), we essentially only need to scale the computed \([y_i] = [\frac{1}{b}]\) with \([c]\) (as \(\frac{1}{b} \cdot c = \frac{1}{bc} = \frac{1}{2}\)). 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 \(a = a'(\alpha)\) and output \(b = a \cdot c = b'(\ell)\) of MSNZBFit, we have

\[
y_i = \frac{2^\ell}{b'} = \frac{2^\ell}{a'(\ell) \cdot c} \cdot 2^{\ell-\alpha} = \frac{1}{2} \cdot 2^{2\ell-\alpha}
\]

and that the output should be scaled with \(2^\delta\).

We define the corresponding functionality \(F_{inv}\) 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 \(F_{inv}\) in the \((F_{msnzb}, F_{div}, F_{mult})\)-hybrid model in the presence of a passive adversary.
Protocol 5 Integer Division with Private Divisor

Functionality: \([\bar{z}] \leftarrow \text{Div}_\text{priv}(\bar{a}, \bar{d})\)
Input: \([\bar{a}], [\bar{d}]\), where \(a = a_{(\alpha)} \in \mathbb{Q}_{(\ell,\alpha)}\), and \(d = d_{(\beta)} \in \mathbb{Q}_{(\ell,\beta)}\)
Output: \([\bar{z}]\), where \(z \approx (\frac{\bar{z}}{\bar{d}})_{(\delta)}\)
Parameter: \((\ell, I, \alpha, \beta, \delta)\) where \(\alpha\) and \(\beta\) are the offsets of \(a\) and \(d\), respectively.
1: \([\bar{z}'] \leftarrow \text{Inv}_{(\ell, I, \alpha, \beta, \delta)}([\bar{d}]) \quad \triangleright \quad \bar{z}' = (\frac{\bar{z}}{\bar{d}})_{(\delta)}
2: \text{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 \([\bar{a}]\) and \([\bar{d}]\), we simply compute \([\frac{1}{\bar{d}}]\) using Inv, and multiply this with \([\bar{a}]\) to obtain \([\frac{\bar{a}}{\bar{d}}]\). The resulting protocol, Div\_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 \(\ell\) bits of precision for the inversion, which ensures the result is equal to \(\frac{\bar{a}}{\bar{d}}\) for \(\ell\)-bit fixed-point values. We define the corresponding functionality \(F_{\text{Div}_\text{priv}}\) that as on input \([\bar{a}]\) and \([\bar{d}]\) outputs \([\bar{a}] \cdot \bar{d}^{-1}\) in which \([\frac{1}{\bar{d}}]\) is obtained by \(F_{\text{Inv}}\).

Theorem 5: The protocol Div\_priv securely computes fixed-point division \(F_{\text{Div}_\text{priv}}\) in the \((F_{\text{Inv}}, F_{\text{div}}, F_{\text{mult}})\)-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 \(y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = y_n(3 - x \cdot y_n^2)\) 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 \(x = b \cdot 2^e\) for \(b \in \left[\frac{1}{2}; 1\right)\), which implies \(1 \approx \frac{1}{\sqrt{b}} = \left(\frac{1}{\sqrt{2}}\right) \cdot 2^{-e/2}\) if \(e\) is even
\[
\frac{1}{\sqrt{b}} = \left(\frac{1}{\sqrt{2}}\right) \cdot 2^{-e/2} \quad \text{if } e \text{ is even}
\]
\[
\frac{1}{\sqrt{b}} = \left(\frac{1}{\sqrt{2}}\right) \cdot 2^{-(e+1)/2} \quad \text{if } e \text{ is odd}
\]
Hence, we only need to compute \(\frac{1}{\sqrt{2}}\) for \(b \in \left[\frac{1}{2}; 1\right)\), 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, 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 MSNZBFit, the extended MSNZBFitExt right-shifts the input \(a\) to obtain an \(\ell\)-bit value \(b = a'c\), that when interpreted as an element of \(\mathbb{Q}_{(\ell,\ell)}\), represents \(b \in \left[\frac{1}{2}; 1\right)\), and that \(c' = \sqrt{c}\). Having \(r\) and \(c'\) allows us to compute \(\frac{1}{\sqrt{b}}\) as \(\frac{1}{\sqrt{2}} \cdot (1 + r \cdot (\sqrt{2} - 1)) \cdot c'\). Finally, note 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 \(2^{\delta - \frac{3}{2}\ell + \frac{\delta}{2}}\), note that for input \(a = a'c\) and output \(b = a'c' = b' \cdot 2^e\) of MSNZBFitExt, we have
\[
y = 2^e \sqrt{b} = 2^e \frac{2^\ell}{\sqrt{a'c'c^2-2\ell}} = \frac{1}{\sqrt{a'c'c^2-2\ell}} \cdot 2^{\delta - \frac{3}{2}\ell + \frac{\delta}{2}}
\]
and that the output should be scaled with \(2^\delta\). We define the functionality \(F_{\text{InvSqrt}}\) that computes \(\frac{1}{\sqrt{a}}\) as done in the above using the Newton’s method.

Theorem 6: The protocol InvSqrt securely computes the inverse of the square root functionality \(F_{\text{InvSqrt}}\) in the \((F_{\text{multfit}}, F_{\Mod}, F_{\text{div}}, F_{\text{mult}})\)-hybrid model in the presence of a passive adversary.

Given the above protocol InvSqrt for computing \(\frac{1}{\sqrt{x}}\), and noting that \(\sqrt{x} = \frac{1}{\sqrt{2}}\), we can easily construct a protocol for computing \(\sqrt{x}\), simply by running InvSqrt and multiplying the result with \(x\). The resulting protocol, Sqrt, is shown in Protocol 8. Let \(F_{\text{sqrt}}\) be the functionality that on input \([a]\) outputs \([\sqrt{a}]\) in which \([\sqrt{a}]\) is obtained by \(F_{\text{InvSqrt}}\).

Theorem 7: The protocol Sqrt securely computes the square root functionality \(F_{\text{sqrt}}\) in the \((F_{\text{InvSqrt}}, F_{\text{div}}, F_{\text{mult}})\)-
Protocol 8 Square Root

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

1: \([z']\) \leftarrow \text{InvSqrt}(\ell, \alpha, \delta)(\langle a \rangle)\) \quad \triangleright \quad z' = \left(\frac{1}{\sqrt{\alpha}}\right)_{(\delta)}
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) \(\mu\): a lower bound for the input
2) \(b_t, \ldots, b_{t-I}\): bit representation of the \(t\) most significant bits of \(b := a - \mu\)
3) \(b_s\): integer representation of \((a - \mu) - \sum_{t-I \leq i \leq t} 2^{i-\alpha} b_i\)

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

\[
\exp a = \left(\prod_{i=0}^{t-I} \exp(b_i \cdot 2^{i-\alpha})\right)\cdot \exp(b_s)\cdot \exp(\mu).
\]

(6)

Here, \(\alpha\) 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 \(\mu\) is public), \(\prod_{i=t-I}^{t-1} \exp(b_i \cdot 2^i)\) using table lookups, and \(\exp(b_s)\) using the Taylor series. The Taylor series rapidly converge since \(b_s\) is made small due to the subtraction of \(\mu\) 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 \([b]\) 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 \(\prod_{i=t-I}^{t-1} \exp(b_i \cdot 2^i)\). However, to maintain high accuracy, the parties will not use \(\exp 2^i\) directly, but precompute a mantissa \(f_i\) and exponent \(2^i\); such that \(f_i \cdot 2^i = \exp 2^{i-\alpha}\). This allows the parties to compute the product of \(f_i\) values separately from the product of \(2^i\) 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: \([z] \leftarrow \text{Exp}(\langle a \rangle)\)
Input: \([a]\), where \(a = a_{(\alpha)} \in \mathbb{Q}(\ell, \alpha)\)
Output: \([z]\), where \(z \approx \exp(a)\)
Parameter: \((\ell, I, \alpha, \beta, \delta, \mu, t)\) where \(I\) is the number of iterations used in the computation, \(\beta\) is the offset of the lookup table values, \(t\) indicates the lookup table vs Taylor series threshold, and \(\mu\) is a lower bound for the input.

1: \([b] := [a] - \mu(\alpha)\)
2: \([b_t, \ldots, b_{t-I}] \leftarrow \mathcal{F}_\text{BDC}([b])\) \quad \triangleright \quad \text{We only use } \ell - t \text{ MSBs while } \mathcal{F}_\text{BDC} \text{ outputs } \ell \text{ bits.}
3: \([b_{z}] \leftarrow \mathcal{F}_\text{mod}([b_{t}] \langle \ell, \alpha \rangle)\) for \(i = \ell, \ldots, \ell - t\).
4: \([b_{\sigma}] := [b] - \sum_{\ell-I \leq i \leq \ell} 2^i[b_i]\) \quad \triangleright \quad \text{Value of } \ell \text{ LSBs of } [b]
5: \text{Parties define } f_i, \epsilon_i \text{ such that } \exp 2^{i-\alpha} = f_i \cdot 2^{\epsilon_i} \quad \triangleright \quad \text{Precomputed lookup table values}
6: \text{Using } ([f_i]) \leftarrow \mathcal{F}_\text{CondAssign}(1_{(\beta)}, (f_i)_{(\beta)}, [b_i]), \text{ the parties obtain}
7: \text{Using } ([\epsilon_i]) \leftarrow \mathcal{F}_\text{CondAssign}(1_{(2^{\epsilon_i})}, [b_i]), \text{ the parties obtain}
8: \langle f^\prime \rangle := \prod_{i=\ell}^{\ell-I} f_i \cdot 2^{\epsilon_i}
9: \langle \epsilon^\prime \rangle := \sum_{i=\ell-I}^{\ell-I} \epsilon_i
10: b_{\sigma,0} := 1
11: \text{for } i = 1 \text{ to } I - 1 \text{ do}
12: \quad \langle b_{\sigma, i} \rangle \leftarrow \langle b_{\sigma, i-1} \rangle \times \langle b_{\sigma} \rangle
13: \quad \langle b_{\sigma} \rangle := \prod_{0 \leq i < \ell} b_{\sigma, i} \quad \triangleright \quad \text{Division using Div}
14: \text{Output } ([f^\prime]) \cdot \langle \epsilon^\prime \rangle \cdot \langle b_{\sigma} \rangle \cdot \exp(\mu \cdot 2^{\delta-\beta-\alpha})

Lastly, the result is computed as \(\prod_{i=\ell}^{\ell-I} f_i \cdot 2^{\epsilon_i} \cdot \exp a_{\sigma}\). 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 \(\mathcal{F}_\text{BDC}, \mathcal{F}_\text{mod}, \text{ and } \mathcal{F}_\text{div}\) 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

$$x_i \leftarrow \frac{x_i - \mu}{\sigma \sqrt{\epsilon + t}}$$

where $\mu$ and $\sigma$ are mean and variance of $\overrightarrow{x}$, and $\epsilon$ 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 ReLU$(u) = \max(0, u)$. The function 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 softmax$(u_i)$ at the output layer. The softmax function for classification into $k$ classes is as follows:

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

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

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

where $t$ indicates the iteration number of the learning process, $W, \tilde{V}$, and $\tilde{M}$ are matrices, $\circ$ denotes the element-wise multiplication, and $\eta$ and $\epsilon$ 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., $\alpha$ 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 $\mathcal{F}_{BC}$, $\mathcal{F}_{mod}$, and $\mathcal{F}_{QT}$ 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 $\kappa = 8$. All experiments are run in the

<table>
<thead>
<tr>
<th>Table III: Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS \hspace{0.5cm} CentOS Linux release 7.3.1611</td>
</tr>
<tr>
<td>CPU \hspace{0.5cm} Intel Xeon Gold 6144k (3.50GHz 8 core/16 thread) x 2</td>
</tr>
<tr>
<td>Memory \hspace{0.5cm} 768 GB</td>
</tr>
<tr>
<td>SW \hspace{0.5cm} Intel X710/X557-AT 10G Ring configuration</td>
</tr>
</tbody>
</table>
functions. We implement similar elementary functions, but replaces these with MPC-friendly 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 $\epsilon$).

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.

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.
The results are illustrated in Figure 1. For AlexNet, we see that accuracy converges towards 75% ~ 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 ~ 14 for AlexNet and 46 ~ 48 for VGG16, whereas in the WAN setting, the factors are about 2 and 6, respectively.

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. However, including the offline phase, which is significant for Trident, will add considerably to the running time. This strongly indicates that, despite being a four-party protocol, Trident offers worse overall performance compared to our three-party protocols, and is hence not directly comparable to our framework.

\footnote{From the measurements reported in [47], we estimate that the online phase of FALCON in their environment would require 306s and 3026s 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.}

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

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

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

<table>
<thead>
<tr>
<th>Security Setting</th>
<th>AlexNet [h]</th>
<th>VGG16 [h]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALCON Passive LAN</td>
<td>324</td>
<td>3,342</td>
</tr>
<tr>
<td>Ours Passive LAN</td>
<td>22</td>
<td>72</td>
</tr>
<tr>
<td>FALCON Active LAN</td>
<td>1,235</td>
<td>13,108</td>
</tr>
<tr>
<td>Ours Active LAN</td>
<td>104</td>
<td>269</td>
</tr>
<tr>
<td>FALCON Passive WAN</td>
<td>698</td>
<td>3,678</td>
</tr>
<tr>
<td>Ours Passive WAN</td>
<td>346</td>
<td>580</td>
</tr>
<tr>
<td>FALCON Active WAN</td>
<td>2,204</td>
<td>14,914</td>
</tr>
<tr>
<td>Ours Active WAN</td>
<td>1,110</td>
<td>2,155</td>
</tr>
</tbody>
</table>

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. However, including the offline phase, which is significant for Trident, will add considerably to the running time. This strongly indicates that, despite being a four-party protocol, Trident offers worse overall performance compared to our three-party protocols, and is hence not directly comparable to our framework.

\footnote{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


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


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

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

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

**FUNCTIONALITY 9 (F_{\text{BDC}} – Bit decomposition):**

Upon receiving \([a], F_{\text{BDC}}\) reconstructs \(a\), generates shares \(([a]_1, \ldots, [a]_\ell)\), where \(a = \sum_{i=1}^{\ell} 2^{i-1} a_i\), and sends \(([a]_1, \ldots, [a]_\ell)\) to \(P_1\).

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

**FUNCTIONALITY 10 (F_{\text{BDC}} – Bit composition):**

Upon receiving \([a], \ldots, [a], F_{\text{BDC}}\) reconstructs \(a_1, \ldots, a_\ell\), computes \(a := \sum_{i=1}^{\ell} a_i \cdot 2^i\), generates shares \([a]\), and sends \([a]\) to \(P_1\).

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

**FUNCTIONALITY 11 (F_{\text{mod}} – Modulus conversion):**

Upon receiving \([a], F_{\text{mod}}\) reconstructs \(a\), generates shares \([a]\), and sends \([a]\) to \(P_1\).

**APPENDIX B**

**QUOTIENT TRANSFER PROTOCOL**

We describe the quotient transfer protocols for \(\llbracket \cdot \rrbracket\), \(\langle \cdot \rangle\), and \([\cdot]\).

**Protocol 10 Quotient Transfer for \(\langle \cdot \rangle\)**

**Functionality:** \(\langle q \rangle \leftarrow \text{QT}(\langle a \rangle)\)

**Input:** \(\langle a \rangle\) where \(a\) is a multiple of 2.

**Output:** \(\langle q \rangle\) where \(\langle a \rangle + \langle q \rangle = a + qp\) in modulo 2, respectively. Let them be \([\langle a \rangle_1]\) and \([\langle a \rangle_2]\)

\[\begin{align*}
1: & P_0 \text{ and } P_1 \text{ secret-share } \langle a \rangle_1 \text{ and } \langle a \rangle_2 \text{ in modulo } 2, \text{ respectively. Let them be } [\langle a \rangle_1] \text{ and } [\langle a \rangle_2]. \\
2: & [q] := \langle [a]_1 \rangle + \langle [a]_2 \rangle. \\
3: & [q] \leftarrow F_{\text{mod}}([q]) \\
4: & \langle q \rangle \leftarrow \text{ConvertToAdd}([q]) \\
5: & \text{Output } \langle q \rangle
\end{align*}\]

**Protocol 11 Quotient Transfer for \([\cdot]\)**

**Functionality:** \([q] \leftarrow \text{QT}([a])\)

**Input:** \([a]\) where \(a\) is a multiple of 4.

**Output:** \([q]\) where \(a_1 + a_2 + a_3 = a + qp\)

\[\begin{align*}
1: & P_0 \text{ and } P_1 \text{ locally generate shares of the second LSBS of } a_1, a_2, \text{ and } a_3 \text{ in modulo } 2, \text{ respectively. Let them be } [a_1^{(1)}], [a_2^{(1)}], [a_2^{(2)}], [a_3^{(1)}], \text{ and } [a_3^{(2)}]. \\
2: & [q_1] := [a_1^{(1)}] + [a_2^{(1)}] + [a_3^{(1)}]. \\
3: & [q] := ([a_1^{(1)}] + [a_2^{(1)}] + [a_3^{(1)}]) + [a_3^{(2)}] + [a_3^{(1)}]. \\
4: & [q_1] \leftarrow F_{\text{mod}}([q]) \\
5: & [q_2] \leftarrow F_{\text{mod}}([q]) \\
7: & \text{Output } [q_2] := [q_1] + 2[q_2]
\end{align*}\]

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 multiplications and modulus-conversion protocols. In the protocol, we describe them as functionalities \(F_{\text{mult}}\) and \(F_{\text{mod}}\). Please see [28] for their instantiations.

In the presence of passive adversaries, we use \(\langle a \rangle\) as an input of the quotient transfer protocol. Because \(\langle a \rangle\) 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 \((a_1 \oplus a_3)(a_2 \oplus a_3) \oplus 3\). 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, \(F_{\text{div}}\) and \(F_{\text{div}, \text{general}}\), corresponding to division in the specific and general case, respectively, in Functionality 12 and Functionality 13.
FUNCTIONALITY 12 ($\mathcal{F}_{\text{div}}$ – 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 $\left( \left( \langle\langle a \rangle\rangle_1 \leq a \right) \land \left( r_a < r_1 \right) \right) \lor \left( \left( a < \langle\langle a \rangle\rangle_1 \right) \land \left( r_a - 1 < r_1 \right) \right)$.
- $b = a/d + 1$ 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 ($\mathcal{F}_{\text{div,general}}$ – Division by a public value in general case):
Upon receiving $\langle\langle a \rangle\rangle_1$ and $\langle\langle a \rangle\rangle_2$, $\mathcal{F}_{\text{div,general}}$ 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, $\mathcal{F}_{\text{div,general}}$ sets $b$ as follows:
- $b = a/d$ if $\left( \left( \langle\langle a \rangle\rangle_1 \leq a \right) \land \left( r_a < r_1 \leq r_p \right) \right) \lor \left( \left( a < \langle\langle a \rangle\rangle_1 \right) \land \left( r_a < r_1 \land \left( (r_a + r_p) < r_1 \lor r_a > r_1 \lor r_1 < r_p \right) \right) \lor \left( \left( a < \langle\langle a \rangle\rangle_1 \right) \land \left( r_p < r_1 \leq r_a + r_p \right) \right)$.
- $b = a/d + 1$ if $\left( \left( \langle\langle a \rangle\rangle_1 \leq a \right) \land \left( r_a < r_1 \leq r_p \right) \right) \lor \left( \left( a < \langle\langle a \rangle\rangle_1 \right) \land \left( r_p < r_1 \leq r_a + r_p \right) \right)$.
- $b = a/d + 2$ if $\left( \left( \langle\langle a \rangle\rangle_1 \leq a \right) \land \left( (r_a + r_p) < r_1 \right) \right)$.
Then, $\mathcal{F}_{\text{div,general}}$ 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 $\lceil \cdot \rceil$

Functionality: $[q] \leftarrow \text{QT}(\langle\langle a \rangle\rangle)$
Input: $[a]$ (no restriction of $a$)
Output: $[q]$ where $a_1 + a_2 + a_3 = a + 2q$
1: $F_0$ and $P_1$ locally generate shares of $a_1$, $a_2$, and $a_3$ in modulo $2$, respectively. Let them be $\langle\langle a_1 \rangle\rangle$, $\langle\langle a_2 \rangle\rangle$, and $\langle\langle a_3 \rangle\rangle$.
2: $[q] \leftarrow \langle\langle a_1 \rangle\rangle \oplus \langle\langle a_3 \rangle\rangle \cdot \langle\langle a_2 \rangle\rangle \oplus \langle\langle a_3 \rangle\rangle$.
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 $\lceil \cdot \rceil$ with Signed Integers

Functionality: $[c] \leftarrow \text{Div}^S_{\langle\langle a \rangle\rangle}(\langle\langle a \rangle\rangle, d)$
Input: Share of dividend $\langle\langle a \rangle\rangle$ and (public) divisor $d$, where $2^{|a|} - r_a \leq a \leq 2^{|a|} - 2 - r_a - 1$
Output: $[c]$, where $c \approx q$
Parameter: $\omega := \frac{2^{|a| - 2} - r_a}{2^{|a| - 2} - 1}$ and $r_a$ such that $\omega d = 2^{|a| - 2} + r_a$
1: $[b] \leftarrow \text{Div}^S_{\langle\langle a \rangle\rangle}(\langle\langle a \rangle\rangle + \omega, d)$
2: $[c] \leftarrow [b] - \langle\langle a \rangle\rangle$
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
\[\alpha_a - q + 1 + \frac{r_a + q(d - 1) - r_1 - r_2}{d} = \alpha_a - q + 1 + \frac{r_a + q(d - 1) - r_1 - r_2}{d}\]
\[\text{Eq. (10)}\]

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
\[r_a - r_1 - r_2 = \frac{\alpha_a}{d}, r_a + r_p - r_1 = \frac{\alpha_a + \alpha_2}{d}\]
Since $\alpha_1$, $\alpha_2$, and $\alpha_a$ 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 $-d$.

Therefore, in the case of $q = 0$, the output is as follows.
\[\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d} = \begin{cases} \alpha_a & \text{if } r_a < r_1 \\ \alpha_a + 1 & \text{otherwise} \end{cases}\]
\[\text{Eq. (11)}\]

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 $q = 1$, $(\alpha_1 d + r_1) + (\alpha_2 d + r_2) = (\alpha_a d + r_1) + (\alpha_p d + r_1)$ and
\[r_a + r_p - r_1 - r_2 = \alpha_1 + \alpha_2 - (\alpha_a + \alpha_p)\]
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

$$b = \begin{cases} \frac{a}{d} & \text{if } \left( (\{a\} \leq a) \land (r_a < r_1) \right) \\
\frac{a}{d} + 1 & \text{otherwise} \end{cases} \quad (13)$$

### B. General Case

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

We first consider the $q = 0$ case. In this case, Eq. (11) is

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

For the term $\frac{r_a - r_1 - r_2}{d}$, the same discussion in the specific case holds. We then consider the last term $(r_1 - r_p + d - 1)/d$. If $r_1 - r_p \leq 0$, $0 \leq r_1 - r_p + d - 1 < d$ since $-d + 1 \leq r_1 - r_p$. Otherwise, $d \leq r_1 - r_p + d - 1 < 2d$ since $r_1 - r_p \leq d - 1$. Therefore,

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

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

$$\alpha_a + 1 + \frac{r_a - r_1 - r_2}{d} + (r_1 + d - 1 - r_p)/d = \begin{cases} \alpha_a & \text{if } r_a < r_1 \leq r_p \\ \alpha_a + 1 & \text{if } (r_1 \leq r_a) \land (r_1 \leq r_p) \\ \alpha_a + 2 & \text{if } r_p < r_1 \leq r_a \end{cases} \quad (15)$$

Next, we consider the $q = 1$ case. In this case, Eq. (11) 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 $\frac{r_a + r_p - r_1 - r_2}{d} \text{ must be a multiple of } d$. 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,

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

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

$$\alpha_a + \frac{r_a + r_p - r_1 - r_2}{d} + (r_1 - r_p + d - 1)/d = \begin{cases} \alpha_a & \text{if } (r_a + r_p < r_1) \lor (r_a + r_p - d < r_1 \leq r_p) \\ \alpha_a + 1 & \text{otherwise} \end{cases} \quad (17)$$

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

$$\left( (\{a\} \leq a) \land (r_a < r_1 \leq r_p) \right) \lor$$

$$\left( (a < \{a\} \land (r_a + r_p < r_1) \lor (r_a + r_p - d < r_1 \leq r_p)) \right),$$

$$b = a/d + 1$$

if

$$\left( (\{a\} \leq a) \land (r_1 \leq r_a) \land (r_1 \leq r_p) \right) \lor$$

$$\left( (a < \{a\} \land (r_1 \leq r_a + r_p) \lor (r_1 \leq r_a + r_p - d)) \right),$$

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 $d$, $a$, $p$, $\{a\}$ (and their dependent variables $\alpha_a$, $r_a$, $\alpha_p$, $r_p$, $\{a\}_2$, $r_1$, and $r_2$). In this section, we assume that $\{a\}_1$ 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 satisfy $(\{a\}_1 \leq a) \land (r_a < r_1)$ or $(a < \{a\}_1) \land (r_a - 1 < r_1)$. We observe that $(\{a\}_1 \leq a) \land (r_a < r_1)$ is true if $\{a\}_1 \in \{md + n \mid 0 \leq m \leq \alpha_a - 1, r_a + 1 \leq n \leq d - 1\}$. The number of integers satisfying the above is $\alpha_a(d - r_a - 1)$.

We then observe that $(a < \{a\}_1) \land (r_a - 1 < r_1)$ is true if $\{a\}_1 \in \{md + n \mid \alpha_a \leq m \leq \alpha_a - \alpha_p, r_a \leq n \leq d - 1\}$

\{\alpha_a \text{d} + 1 \} \text{d} = \alpha_a(d - r_a - 1) - 2

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

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

$$\frac{\alpha_a(d - r_a - 1) + (\alpha_a - \alpha_p + 1)(d - r_a) - 2}{p}$$

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

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

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

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

#### 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., \( \langle a \rangle_1 \leq a \). If \( r_p < r_a \), Eq. (15) equals to

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

\[
\begin{cases}
\alpha_a + 1 & \text{if } (r_1 \leq r_p) \lor (r_a < r_1) \\
\alpha_a + 2 & \text{if } r_p < r_a \leq r_a + r_p
\end{cases}
\]

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

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

and

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

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

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

\[
\begin{cases}
\alpha_a & \text{if } (r_a + r_p < r_1) \lor (r_1 \leq r_p) \\
\alpha_a + 1 & \text{if } (r_p < r_1 \leq r_a + r_p)
\end{cases}
\]

Therefore, if \( r_p < r_1 \),

\[
\Pr[(q = 1) \land ((r_a + r_p < r_1) \lor (r_1 \leq r_p))] = \frac{(\alpha_a - \alpha_a)(d - r) - 1}{p}
\]

and

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

In summary, if \( r_p < r_a \) and \( r_a + r_p < d \),

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

and

\[
\Pr[\text{output is } \alpha_a + 1] = \Pr[(q = 0) \land ((r_1 \leq r_p) \lor (r_a < r_1))] \lor (q = 1) \land (r_p < r_1) \lor (r_1 \leq r_p)) = \frac{(d - r_a + r_p)\alpha_a + r_p + 1 + (\alpha_p - \alpha_a - 1)r_a + r_p}{p}
\]

and

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

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

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

\[
\begin{cases}
\alpha_a & \text{if } (r_a + r_p - d < r_1 \leq r_p) \\
\alpha_a + 1 & \text{if } (r_p < r_1) \lor (r_1 \leq r_a + r_p - d)
\end{cases}
\]

If \( r_p < r_a \),

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

and

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

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

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

and

\[
\Pr[\text{output is } \alpha_a + 1] = \Pr[((q = 0) \land ((r_1 \leq r_p) \lor (r < r_1))) \lor (q = 1) \land (r_1 \leq r_a + r_p - d)) = \frac{(d - (r_a - r_p))\alpha_a + r_p + 1 + (\alpha_p - \alpha_a - 1)r_a + r_p}{p}
\]

and

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

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

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

\[
\begin{cases}
\alpha_a & \text{if } r_a < r_1 \leq r_p \\
\alpha_a + 1 & \text{if } (r_1 \leq r_a) \lor (r_p < r_1)
\end{cases}
\]

Similar to the previous case, we have

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

and

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

The \( q = 1 \) case, if \( r_a \leq r_p \),

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

In summary, if \( r_a \leq r_p \) and \( r_a + r_p < d \),
\[ \Pr[\text{output is } \alpha_a] \]
\[ = \Pr\left[ ((q = 0) \land (r < r_1 \leq r_p)) \lor ((q = 1) \land (r_a + r_p < r_1) \lor (r_1 \leq r_p)) \right] \]
\[ = \frac{(r_p - r)\alpha_a + (\alpha_p - \alpha_a)(d - r) - 1 + r_p - r}{p} \]
\[ = \frac{(\alpha_p - \alpha_a)d + r_p\alpha_a - \alpha_p\alpha_a - 1 + r_p - r}{p} \]
\[ = \frac{p - a + r_p\alpha_a - \alpha_p\alpha_a - 1}{p} \]
and
\[ \Pr[\text{output is } \alpha_a + 1] \]
\[ = \Pr\left[ ((q = 0) \land (r_1 \leq r_a) \lor (r_p < r_1)) \lor ((q = 1) \land (r < r_1 \leq r_a + r_p)) \right] \]
\[ = \frac{(d - r_p + r_a)\alpha_a + r_a + 1}{p} + \frac{(\alpha_p - \alpha_a)\alpha_a + r_a - 1 + r_p - r}{p} \]
\[ = \frac{d - r_p\alpha_a + (\alpha_p + 1)\alpha_a + r_a}{p} \]
\[ = \frac{a - r_p\alpha_a + r_a\alpha_a + 1}{p} \]

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 \( r_a \leq r_p \),
\[ \Pr[(q = 1) \land (r_a + r_p - d < r_1 \leq r_p)] \]
\[ = \frac{(\alpha_p - \alpha_a)(d - r_a) - 1 + r_p - r_a}{p} \]
and
\[ \Pr[(q = 1) \land ((r_p < r_1) \lor (r_1 \leq r_a + r_p - d))] \]
\[ = \frac{(\alpha_p - \alpha_a)r_a}{p} \]

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 \( a/2^\ell \) 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.

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: $\left[ \sum_{i \leq \ell} 2^i a_i \right] \leftarrow \text{BC}([a_1], \ldots, [a_\ell])$

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

Output: $\left[ \sum_{i \leq \ell} 2^i a_i \right]$

Parameter: The bit-length of secret $\ell$

1. Each $P_i$ sets $[b_i] := [0]$ $\triangleright$ Set its sub-share as $(0, 0)$
2. $[a_i'] := [a_i]$
3. $[q_i'] \leftarrow F_{\text{QT}}([a_i'])$
4. for $i := 2$ to $\ell$ do
5. $[a_i'] := [a_i] - [q_{i-1}] - [b_{i-1}]$
6. $[b_i] := ((1 - [a_i]) + [b_{i-1}]) \cdot ([q_i] - [b_{i-1}]) + [b_{i-1}]$ $\triangleright$ The known bit-composition protocol in [2] cancel $2$ into $\ell$ bit
7. $[q_i] \leftarrow F_{\text{QT}}([a_i'])$
8. $[b_i] \leftarrow F_{\text{mod}}([b_i])$
9. $[q_i] \leftarrow F_{\text{mod}}([q_i'])$
10. $[a_i'] \leftarrow \sum_{i \leq \ell} 2^i [a_i'_j] \mod p$
11. Output $[a_i'] - 2^\ell ([b_i] + [q_i'])$

In the bit-composition protocol, we want to obtain the composed value $[a_i]$ on input of its binary representation $[a_1], \ldots, [a_\ell]$ by computing addition as $\sum 2^i [a_i]$. However, this computation does not work since each $[a_i]$ has the quotient $q_i$ so $2^i [a_i] = 2^i [q_i] + 2^{i-1} a_i$ 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 $i \leq \ell - 1$, and they does not need to obtain $q_i$ for $\ell > i$: their underlying secret sharing is on modulus $2^\ell$ so $2^i q_i$ 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 $F_{\text{QT}}$ and $F_{\text{mod}}$ by Protocol 12 and that in [28], where $\ell$ is the maximum bit-length of shared secret.

Theorem 15: The protocol in Figure 16 securely implements $F_{\text{BC}}$ in the $(F_{\text{QT}}, F_{\text{mult}}, F_{\text{mod}})$-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 $d_{L=0}$ denotes the number of neurons in layer $j$, and $N_j$ denotes the $i$-th neuron in layer $L = j$. The strength of the coupling of neurons is described by parameters $w_{ij}$. 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 $\text{Func}(A)$, such as $\sqrt{A}$ and $F_{\text{BC}}(A)$.

#### 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, $M, V, M, V$ are matrices of the same size as $G_t$ and $M_t, V_t$ are initialized by 0. Here the superscript $t$ such as $\beta^t$ represents the $t$-th power of $\beta$. Adam proceeds as follows.

$$M_{t+1} = \beta_1 M_{t} + (1 - \beta_1) G_{t}$$
$$V_{t+1} = \beta_2 V_{t} + (1 - \beta_2) G_{t} \circ G_{t}$$
$$M_{t+1} = \frac{1}{1 - \beta_1^t} M_{t+1}$$
$$\hat{V}_{t+1} = \frac{1}{1 - \beta_2^t} V_{t+1}$$
$$\hat{W}_{t+1} = \hat{W}_{t} - \frac{\eta}{\sqrt{\hat{V}_{t+1} + \epsilon}} \circ \hat{M}_{t+1}$$

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: $[Z] \leftarrow \text{ReLU’}([U])$

Input: A matrix $[U]$

Output: $[Z]$, where each element $z_{i,j}$ in $Z$ is 0 if $z_{i,j} \leq 0$ and 1 otherwise

Parameter: $\ell$, where each element in $Z$ is between $2^{-\ell} + 1$ and $2^\ell - 1$.

1. $[U'] = 2^{\ell+1} + [U]$$\triangleright$ Add $2^{\ell+1}$ to each element
2. $([U_{(i+1)^{\ell+1}}], \ldots, [U_{(i)^{\ell+1}}]) \leftarrow F_{\text{BC}}([U])$
3. $[Z] \leftarrow F_{\text{mod}}([U_{(i+1)^{\ell+1}}])$$\triangleright$ Each element in $U_{(i+1)^{\ell+1}}$ is 1 if the corresponding element in $U_{(i)^{\ell+1}}$ is positive and 0 otherwise
4. Output $[Z]$

Protocol 18 Secure ReLU Function

Functionality: $[Y] \leftarrow \text{ReLU}([U])$

Input: A matrix $[U]$

Output: $[Y]$, where $Y \leftarrow \text{ReLU}(U)$

Parameter: $\ell$, where each element in $U$ is between $2^{-\ell} + 1$ and $2^\ell - 1$.

1. $[Z] = \text{ReLU’}([U])$
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.
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 $\text{InvSqrt}(\cdot, \nu)$ to denote computation of the inverse square root with an additional right-shift $\nu$.

4) Secure Feedforward Deep Neural Networks with Adam: Let $n$ be the number of hidden layers, $\beta_{1,t} = \frac{1}{1 - \theta_t}$, and $\beta_{2,t} = \frac{1}{1 - \theta_t^2}$. We show a protocol for secure feedforward deep neural networks with Adam in Protocol 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 convolutional 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 functionalities and notations for this protocol: Let $F_{\text{compare}}$ be the functionality on input $\{\hat{d}^i\} = \{(a_1^i), \ldots, (a_n^i)\}$ and $\{\hat{b}^i\} = \{(b_1^i), \ldots, (b_n^i)\}$ outputs $\{\hat{c}^i\}$, where $c_i = 1$ if $a_i \geq b_i$ and 0 otherwise for $1 \leq i \leq n$. A parallel execution of a comparison protocol, e.g., [28], can realize $F_{\text{compare}}$. Let $F_{\text{CondAssignShare}}$ be the functionality that is similar to $F_{\text{CondAssignShare}}$ but assigned values are not plaintexts but shares: on input of $\{(\hat{d}^i), (\hat{d}^i), (\hat{c}^i)\}$ such that $c_i \in \{0, 1\}$, outputs $\{\hat{d}^i\}$, where $d_i = a_i$ if $c_i = 1$ and $d_i = b_i$ otherwise for $1 \leq i \leq n$. This functionality can be realized in a similar way that computes $c_i \cdot a_i + (1 - c_i) \cdot b_i$ using multiplication protocol in parallel. For two vectors $\{\hat{c}^i\} = \{(c_1^i), \ldots, (c_n^i)\}$ and $\{\hat{f}^i\} = \{(f_1^i), \ldots, (f_n^i)\}$, $\{\hat{c}^i\} || \{\hat{f}^i\}$ denote the concatenation of the vectors, i.e., $\{\hat{c}^i||\hat{f}^i\} = \{(c_1^i), \ldots, (c_n^i), (f_1^i), \ldots, (f_n^i)\}$.

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: \((y] \rightarrow \text{MaxFlag} (\{x\}])\)

Input: A vector of length \(n\), \([x] = ([x_1], \ldots, [x_n])\)

Output: The maximum \([y]\) and its original location \([\bar{y}]\) = \((f_1), \ldots, (f_n)\), where \(y = \max_{1 \leq i \leq n} x_i\) and \(f_j = 1\) if \(y = x_j\) and \(f_j = 0\) otherwise for \(1 \leq j \leq n\).

1: if \(n = 1\) then return \(([x_1], ([1])\).
2: else
3: \(m = n/2\)
4: \([\tilde{x}] := ([x_1], \ldots, [x_m])\)
5: \([\tilde{b}] := ([x_{m+1}], \ldots, [x_n])\)
6: \([\tilde{a}] := \text{FCompute} ([\tilde{x}], [\tilde{b}])\)
7: \([\tilde{d}] := \text{FCondAssignShare} ([\tilde{x}], [\tilde{a}], [\tilde{c}])\)
8: if \(n \mod 2 = 0\) then
9: \((y], [\tilde{c}]) \rightarrow \text{MaxFlag} ([\tilde{d}])
10: \([\bar{y}] \rightarrow \text{Fmath} ([\tilde{a}], [\tilde{c}])\)
11: \([\bar{y}] := [\tilde{a}](\lceil [\tilde{c}] - [\tilde{d}]\rceil)\)
12: else
13: \((y], [\tilde{c}]) \rightarrow \text{MaxFlag} ([\tilde{d}])\)
14: \([\bar{y}] \rightarrow \text{Fmath} ([\tilde{a}], [\tilde{c}])\)
15: \([\bar{y}] := [\tilde{a}](\lceil [\tilde{c}] - [\tilde{d}]\rceil)\)

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 \(\frac{1}{n} \sum \frac{|a_i - c_i|}{c_i}\) and \(\max_i \frac{|a_i - c_i|}{c_i}\), 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 \(\frac{b.335}{2}\) 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 error number refers to the value \(\frac{1}{n} \sum \frac{|a_i - c_i|}{c_i}\) and \(\max_i \frac{|a_i - c_i|}{c_i}\), 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

<table>
<thead>
<tr>
<th></th>
<th>Passive</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truncation</td>
<td>0.304</td>
<td>1.000</td>
</tr>
<tr>
<td>Division</td>
<td>0.335</td>
<td>1.059</td>
</tr>
</tbody>
</table>

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. 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.

8It 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]

<table>
<thead>
<tr>
<th></th>
<th>Passive 1,000 [records]</th>
<th>Passive 1,000,000</th>
<th>Active 1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Truncation</td>
<td>0.612</td>
<td>14.80</td>
<td>4.51</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.0219</td>
<td>1.179</td>
<td>0.236</td>
</tr>
<tr>
<td>Square root</td>
<td>0.0147</td>
<td>0.608</td>
<td>0.136</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.0355</td>
<td>1.017</td>
<td>0.237</td>
</tr>
</tbody>
</table>

[48]

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

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time [s]</td>
<td>225</td>
<td>117</td>
<td>64</td>
<td>37</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th># of Neurons</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>68</td>
<td>85</td>
<td>117</td>
<td>304</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th># of Hidden Layers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>102</td>
<td>117</td>
<td>133</td>
<td>149</td>
</tr>
</tbody>
</table>

Large Data: 100 Features × 10 Million Data Samples.
Table XIII shows the execution time per epoch using 100 attributes × 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]

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Our Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>4,047 [s] (1.12 [h])</td>
</tr>
<tr>
<td>1024</td>
<td>2,724 [s] (0.76 [h])</td>
</tr>
</tbody>
</table>