ABY^3: A Mixed Protocol Framework for Machine Learning

Payman Mohassel and Peter Rindal

Abstract

Machine learning is widely used to produce models for a range of applications and is increasingly offered as a service by major technology companies. However, the required massive data collection raises privacy concerns during both training and prediction stages.

In this paper, we design and implement a general framework for privacy-preserving machine learning and use it to obtain new solutions for training linear regression, logistic regression and neural network models. Our protocols are in a three-server model wherein data owners secret share their data among three servers who train and evaluate models on the joint data using three-party computation (3PC).

Our main contribution is a new and complete framework (ABY^3) for efficiently switching back and forth between arithmetic, binary, and Yao 3PC which is of independent interest. Many of the conversions are based on new techniques that are designed and optimized for the first time in this paper. We also propose new techniques for fixed-point multiplication of shared decimal values that extends beyond the three-party case, and customized protocols for evaluating piecewise polynomial functions. We design variants of each building block that is secure against *malicious adversaries* who deviate arbitrarily.

We implement our system in C++. Our protocols are up to *four orders of magnitude* faster than the best prior work, hence significantly reducing the gap between privacy-preserving and plaintext training.

1 Introduction

Machine learning is widely used to produce models that classify images, authenticate biometric information, recommend products, choose which Ads to show, and identify fraudulent transactions. Major technology companies such as Microsoft, IBM, Amazon, and Google are providing cloud-based machine learning services [4, 5, 7, 2] to their customers both in form of pre-trained models that can be used for prediction as well as training platforms that train models on customer data. Advances in *deep learning*, in particular, have lead to breakthroughs in image, speech, and text recognition to the extent that the best records are often held by neural network models trained on large datasets.

A major enabler of this success is the large-scale data collection that deep learning algorithms thrive on. Internet companies regularly collect users' online activities and browsing behavior to train more accurate recommender systems, the healthcare sector envisions a future where patients' clinical and genomic data can be used to produce new diagnostic models and there are efforts to share security incidents and threat data, to improve future attack prediction.

The data being classified or used for training is often sensitive and may come from multiple sources with different privacy requirements. Regulations such as HIPPA, PCI, and GDPR, user privacy concerns, data sovereignty issues, and competitive advantage are all reasons that prevent entities from pooling different data sources to train more accurate models.

Privacy-preserving machine learning based on secure multiparty computation (MPC) is an active area of research that can help address some of these concerns. It ensures that during training, the only information leaked about the data is the final model (or an encrypted version), and during prediction, the only information leaked is the classification label. These are strong guarantees that, though do not provide a full-proof privacy solution (the models themselves or interactions with them can leak information about the data [60, 56, 58]), provide a strong first line of defense which can be strengthened when combined with orthogonal mechanisms such as differential privacy [8, 46]. The most common setting considered in this line of work is a *server-aided* model where data owners (clients) send encrypted version of their data to multiple servers who perform the training procedure on the combined data or apply a (shared) pre-trained model to classify new data samples. Performance of these solutions has improved significantly over the years, leading to orders of magnitude speedup in privacy-preserving machine learning. Nevertheless, there is still a large gap between plaintext training and the privacy-preserving solutions. While part of this gap is unavoidable given the desired guarantees, the current state of affairs is far from optimal. In the three-party computation (3PC) setting with one corruption, following up on a large body of work [13, 14, 23, 41], new techniques and implementations [47, 10, 31] have significantly reduced this gap, e.g. processing 7 billion AND gates per second. The MPC techniques for machine learning, however, are primarily limited to the two-server model and do not benefit from these speedups. They also only consider security against the weaker semi-honest attackers.

In this paper, we explore privacy-preserving machine learning in the three-server model. We emphasize this does not mean only three data owners can participate in the computation. We envision application scenarios where the servers are not considered the same as data owners. Each server can be an independent party or the representative for a subset of data owners. In other words, as long as we guarantee that at most one of the three servers is compromised, an arbitrary number of data owners can incorporate their data into the framework.

A natural question is whether directly applying the new 3PC techniques to machine learning algorithms would yield the same speedup for server-aided privacy-preserving machine learning. Unfortunately, when using existing techniques the answer is negative.

- The first challenge is that the above-mentioned 3PC techniques are only suitable for computation over a $\mathbb{Z}_{2^k}$ ring. This is in contrast with machine learning computation wherein both the training data and the intermediate parameters are decimal values that cannot be natively handled using modular arithmetic. The two most common solutions are to (i) represent decimal values as integers where the least significant bits represent the fractional part, and choose a large enough modulo to avoid a wrap around. This approach fails when performing many floating point multiplications, which is the case in standard training algorithms (e.g. stochastic gradient descent) where millions of sequential multiplications are performed. Moreover, a large modulo implies a more expensive multiplication that further reduces performance, (ii) perform fixed-point multiplication using a boolean multiplication circuit inside the MPC. Such a boolean circuit can be evaluated using either the secret sharing based [10] or the garbled circuit based [47] techniques, leading to a significant increase in either round or communication cost of the solution, respectively.

- The second challenge is that most machine learning procedures require switching back and forth between arithmetic operations such as multiplication and addition, and non-arithmetic operations

such as approximate activation functions (e.g. logistic function), and piecewise polynomial functions (e.g. RELU). The former is most efficiently instantiated using arithmetic secret sharing while the latter should be implemented using either binary secret sharing or Yao sharing. Standard ways of converting between different sharing types are costly and quickly become a major performance bottleneck.

Addressing the above challenges efficiently is even harder in presence of an attacker who behaves arbitrarily malicious.

1.1 Our Contribution

We design and implement a general framework for privacy-preserving machine learning in the three-server model with a single corrupted server. Our contributions are as follows:

1. New approximate fixed-point multiplication protocols for shared decimal numbers at a cost close to a standard secret shared modular multiplication, in both the semi-honest and the malicious case. Experimentally, we find that our protocol results in a 50× improvement in throughput and 24× improvement in latency compared to an optimized boolean circuit. We note that the recent fixed-point multiplication techniques of [48] fail in the 3PC setting and certainly fail in presence of malicious adversaries. Our new techniques are not only secure against malicious adversaries but also extend to arbitrary number of parties.

2. A new general framework for efficiently converting between binary sharing, arithmetic sharing [10] and Yao sharing [47] in the three-party setting, that yields the first Arithmetic-Binary-Yao (ABY) framework for the three-party case with security against malicious adversaries (See Table 1). Many of these conversions are based on new techniques and are designed and optimized for the first time in this paper. Our framework is of general interest given that several recent privacy-preserving machine learning solutions [48, 43, 51] extensively utilize ABY conversions, and its use cases go beyond machine learning [24]. As we will see later, the techniques we develop for our ABY framework are quite different from the original two-party framework of [26], since secure three-party computation techniques deviate significantly from their two-party counterparts.

3. Other optimizations include a *delayed re-share* technique that reduces the communication complexity for vectorized operations by several orders of magnitude and a customized 3PC protocol for evaluating piecewise polynomial functions based on a *generalized three-party oblivious transfer* primitive.

4. We instantiate all our building blocks in both the semi-honest and the malicious setting, often requiring different techniques.

5. We implement our framework in the semi-honest setting and run experiments for training and inference for linear, logistic regression and neural network models. Our solutions are up to 55000× faster than the two-party solution of SecureML [48] when training neural networks, and our framework can do 5089 linear regression training iterations per second compared to 3.7 iterations by [48]. Similarly, our neural network experiment can generate a handwriting prediction in 10 milliseconds compared to the state-of-the-art Chameleon [51] protocol requiring 2700 milliseconds.

1.2 Overview of Techniques

As a brief notational introduction, we define $[\![x]\!]$ as the sharing of a secret value x. This sharing will be one of three types: 1) $[\![x]\!]^{\mathsf{A}}$ denotes an additive secret sharing of $x \in \mathbb{Z}_{2^k}$ over the group $\mathbb{Z}_{2^k}$. 2) $[\![x]\!]^{\mathsf{B}}$ denotes a vector of k binary secret sharing which encodes $x \in \mathbb{Z}_{2^k}$. 3) $[\![x]\!]^{\mathsf{Y}}$ to denote that x is secret shared using keys which are suitable for evaluating a Yao's garbled circuit[47].

Approximate fixed-point multiplication Our starting point is the semi-honest three-party secure computation protocol of Araki et al. [10] based on replicated secret sharing in the ring $\mathbb{Z}_{2^k}$. This protocol represents a value $x \in \mathbb{Z}_{2^k}$ by linearly secret sharing it into three random values $x_1, x_2, x_3 \in \mathbb{Z}_{2^k}$ such that sum of them equals x. Each of the three parties is given two of these shares such that any two parties can reconstruct x. The first challenge in the use of replicated secret sharing is that it does not naturally support fixed-point multiplication and the fixed-point technique introduced in [48] fails in the three-party setting.

We design a new construction for this problem which can be reduced to computing $[\![x]\!] := [\![x'/2^d]\!]$ given $[\![x']\!]$ and d. The solution generates an offline pair $[\![r']\!], [\![r]\!] \in \mathbb{Z}_{2^k}$ where $r = r'/2^d$. Given such a *truncation pair*, parties can truncate a shared value $[\![x']\!]$ by first revealing $x' - r'$ to all and jointly computing $[\![x]\!] = [\![r]\!] + (x' - r')/2^d$. We show that with high probability, x is a correct truncation of x' with at most 1 bit of error in the least significant bit. We also show how to efficiently generate the pair $[\![r]\!], [\![r']\!]$ using high throughput techniques. This approach can be made secure against malicious adversaries and it is easy to see that it generalizes to an arbitrary number of parties.

Moreover, we show that fixed-point multiplication can be further optimized when working with vectors and matrices. In particular, the inner product of two n-dimensional vectors can be performed using $O(1)$ communication and a single truncation pair.

Three-party ABY framework For training linear regression models, we only need to use arithmetic sharing, i.e. additive replicated sharing over $\mathbb{Z}_{2^k}$ where k is a large value such as 64. In logistic regression and neural network training, however, we also need to perform computation that requires bit-level operations. The best way to perform such tasks is to either use binary sharing i.e. additive sharing over $\mathbb{Z}_2$ or Yao sharing based on three-party garbling [47]. The former is more communication efficient, with only $O(n)$ bits communicated for a circuit with n gates, but with the number of rounds proportional to the circuit depth, while the latter only requires 1 or 2 rounds but a higher communication cost.

We show efficient conversions between all three sharing types, with the goal of minimizing both round and communication cost. Please refer to Table 1 for a complete list of our conversion protocols and their cost. When compared to the original two-party ABY framework of [26], we reiterate that our conversion techniques, while functionally similar, differ significantly due to large deviations between 3PC and the less efficient 2PC techniques. To provide a flavor of our techniques, we review our new solution for converting an arithmetic share at the cost of a single addition circuit. Consider $[\![x]\!]^{\mathsf{A}} = (x_1, x_2, x_3)$ where $x = x_1 + x_2 + x_3$. Since we use replicated sharing, party 1 holds both x_1 and x_2 and can compute $x_1 + x_2$ locally. Party 1 then inputs $(x_1 + x_2)$ while party 3 inputs x_3 to a binary sharing 3PC that computes an addition circuit that computes $[\![(x_1 + x_2)]\!]^{\mathsf{B}} + [\![x_3]\!]^{\mathsf{B}}$. Parties also locally compute binary sharing of two random values $[\![y_2]\!]^{\mathsf{B}}, [\![y_3]\!]^{\mathsf{B}}$ which are revealed to parties (1,2) and parties (2,3) respectively. They then locally compute $[\![y_1]\!]^{\mathsf{B}} = ([\![(x_1 + x_2)]\!]^{\mathsf{B}} + [\![x_3]\!]^{\mathsf{B}}) \oplus [\![y_2]\!]^{\mathsf{B}} \oplus [\![y_3]\!]^{\mathsf{B}}$ and reveal it to parties (1,3). This completes the semi-honest

Conversion	Semi-honest		Malicious	
	Comm.	Rounds	Comm.	Rounds
$[\![x]\!]^{\mathsf{A}} \to [\![x]\!]^{\mathsf{B}}$	$k + k \log k$	$1 + \log k$	$k + k \log k$	$1 + \log k$
$([\![x]\!]^{\mathsf{A}}, i) \to [\![x[i]]\!]^{\mathsf{B}}$	k	$1 + \log k$	$2k$	$1 + \log k$
$[\![x]\!]^{\mathsf{B}} \to [\![x]\!]^{\mathsf{A}}$	$k + k \log k$	$1 + \log k$	$k + \log k$	$1 + \log k$
$[\![b]\!]^{\mathsf{B}} \to [\![b]\!]^{\mathsf{A}}$	$2k$	1	$2k$	2
$[\![b]\!]^{\mathsf{Y}} \to [\![b]\!]^{\mathsf{B}}$	$1/3$	1	$2\kappa/3$	1
$[\![b]\!]^{\mathsf{B}} \to [\![b]\!]^{\mathsf{Y}}$	$2\kappa/3$	1	$4\kappa/3$	1
$[\![x]\!]^{\mathsf{Y}} \to [\![x]\!]^{\mathsf{A}}$	$4k\kappa/3$	1	$5k\kappa/3$	1
$[\![x]\!]^{\mathsf{A}} \to [\![x]\!]^{\mathsf{Y}}$	$4k\kappa/3$	1	$8k\kappa/3$	1

Table 1: Conversion costs between arithmetic, binary and Yao representations. Communication (Comm.) is measured in average bits per party. $x \in \mathbb{Z}_{2^k}$ is an arithmetic value, $b \in \{0, 1\}$ is a binary value, κ is the computational security parameter.

conversion to the binary sharing $[\![x]\!]^{\mathsf{B}} = (y_1, y_2, y_3)$. When using a binary sharing 3PC, we use an optimized parallel prefix adder [36] to reduce the number of rounds from k to $log(k)$ at the cost of $O(k \log k)$ bits of communication. For a Yao sharing, we present an optimization which allows the conversion to be performed using k AND gates and *a single* round by leveraging redundancies in the replicated secret sharing.

But this approach is only secure against a semi-honest adversary. A malicious party 1 can use a wrong value in place of $(x_1 + x_2)$ which goes undetected since the addition is done locally. We can prevent this by performing the addition inside another malicious 3PC but this would double both round and communication cost. We introduce new techniques to avoid this extra cost in case of binary sharing 3PC. Consider the traditional ripple-carry full adder where the full adder operation $\mathsf{FA}(x_1[i], x_2[i], c[i-1]) \to (c[i], s[i])$ normally takes two input bits $x_1[i], x_2[i]$ and a carry bit $c[i-1]$ and produces an output bit $s[i]$ and the next carry bit $c[i]$. We modify the full adder to instead take $x_3[i]$ as its third input. It is then easy to see that $x_1 + x_2 + x_3 = 2c + s$. As a result, k parallel invocations of FA in a single round to compute c and s and one evaluation of a parallel prefix adder circuit to compute $2c + s$ are sufficient to compute x. This results in $\log k + 1$ rounds and $k \log k + k$ bits of communication which is almost a factor of 2 better than $2 \log k$ rounds and $2k \log k$ communication for the naive approach. Since the whole computation is done using a 3PC, the resulting protocol is secure against a malicious adversary if the 3PC is.

3PC for piecewise polynomial functions Piecewise polynomial functions compute a different polynomial at each input interval. Activation functions such as RELU are a special case of such functions and many of the proposed approximations for other non-linear functions computed during machine learning training and prediction are also of this form [43, 48]. While our new ABY framework enables efficient three-party evaluation of such functions, we design a more customized solution based on an optimized construction for the following two building blocks: $a[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ (a known by one party) and $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ (a is shared) where b is a bit and $a \in \mathbb{Z}_{2^k}$. We observe that this mixed computation can be instantiated using a generalized three-party oblivious transfer protocol where a bit b_i is the receiver's input and an integer a is the sender's input. We design new protocols for this task with both semi-honest and malicious security that run in 1 and 2 rounds respectively and require between $2k$ to $4k$ bits. This should be of more general interest as piecewise polynomial functions appear in many applications and are a common technique for approximating non-linear functions.

Implementation We implemented our framework in the semi-honest setting and demonstrate that it is faster than all previous protocols. In most cases, this framework improves the overall running time by 100 to 10000× (depending on training model, and problem size) while at the same time reducing the amount of communication. The tasks that we implement include linear, logistic regression and neural network training and evaluation for a variety of problem sizes.

2 Related Work

Earlier work on privacy preserving machine learning considered decision trees [42], k-means clustering [38, 16], SVM classification [63, 61], linear regression [28, 29, 54] and logistic regression [57]. These papers propose solutions based on secure multiparty computation, but appear to incur high efficiency overheads, as they do not take advantage of recent advances in MPC and lack implementation.

Linear Regression Privacy-preserving linear regression in the two-server model was first considered by Nikolaenko et. al. [50] who present a linear regression protocol on horizontally partitioned data using a combination of linearly homomorphic encryption (LHE) and garbled circuits. Gascon et. al. [32] and Giacomelli et. al. [33] extend the results to vertically partitioned data and show improved performance. However, they reduce the problem to solving a linear system using either Yao's garbled circuit protocol or an LHE scheme, which introduces a high overhead and cannot be generalized to non-linear models. In contrast, we use the stochastic gradient descent (SGD) method for training which yields faster protocols and enables training non-linear models such as neural networks. Recent work of Mohassel and Zhang [48] also uses the SGD for training, using a mix of arithmetic, binary, and Yao sharing 2PC (via the ABY framework). They also introduce a novel method for approximate fixed-point multiplication that avoids boolean operations for truncating decimal numbers and yields the state-of-the-art performance for training linear regression models. The above are limited to the two-server model and do not extend to the three-server model considered in this paper. Gilad-Bachrach et. al. [35] propose a framework which supports privacy preserving linear regression. However, the framework does not scale well due to extensive use of garbled circuits.

Logistic Regression Privacy preserving logistic regression is considered by Wu et. al. [62]. They propose to approximate the logistic function using polynomials and train the model using LHE. However, the complexity is exponential in the degree of the approximation polynomial, and as shown in [48] the accuracy of the model is degraded compared to using the logistic function. Aono et. al. [9] consider a different security model where an untrusted server collects and combines the encrypted data from multiple clients, and transfers it to a trusted client to train the model on the plaintext. However, in this setting, the plaintext of the aggregated data is leaked to the client who trains the model.

Neural Networks Privacy preserving machine learning with neural networks is more challenging. Shokri and Shmatikov [55] propose a solution where instead of sharing the data, the two servers share the changes on a portion of the coefficients during the training. Although the system is very efficient (no cryptographic operation is needed at all), the leakage of these coefficient changes is not well-understood and no formal security guarantees are obtained. In addition, each server should be able to perform the training individually

in order to obtain the coefficient changes, which implies each server holds a big portion of a horizontally partitioned data in plaintext.

Privacy-preserving prediction using neural networks models has also been considered in several recent works. In this setting, it is assumed that the neural network is trained on plaintext data and the model is known to one party who evaluates it on private data of another. One recent line of work uses fully homomorphic or somewhat homomorphic encryption to evaluate the model on encrypted data [34, 37, 18, 15]. Another line of work takes advantage of a combination of LHE and garbled circuits to solve this problem [44, 53, 20]. Riazi et al. [51] and Liu et al. [44] each proposes efficiency improvements to the ABY framework and use it for privacy-preserving neural network inference. Chandran et al. [20] propose a framework for automatically compiling programs into ABY components and show how to use it to *evaluate* neural network models. These constructions are all based on two-party protocols and do not benefit from major speed-ups due to new 3PC techniques [10, 31, 47]. They also only provide security against a semi-honest adversary. In Section 7 we give an explicit performance comparison to these frameworks and demonstrate that ours is significantly more efficient.

Very few recent work consider privacy preserving training of Neural Networks. Mohassel and Zhang [48] customize the ABY framework for this purpose and propose a new approximate fixed-point multiplication protocol that avoids binary circuits, and use it to train neural network models. Their fixed-point multiplication technique is limited to 2PC.

To the best of our knowledge, [45] is the only work that considers privacy-preserving machine learning with malicious security and more than two parties. They use the SPDZ library [25] to implement privacy preserving SVM-based image classification and stick to arithmetic operations only and choose a large enough field to avoid overflows during computation. SPDZ provides security against a dishonest majority and hence inherits the same inefficiencies as 2PC. In contrast, we assume an honest majority.

An orthogonal line of work considers the differential privacy of machine learning algorithms [22, 59, 8]. In this setting, the server has full access to the data in plaintext but wants to guarantee that the released model cannot be used to infer the data used during the training. A common technique used in differentially private machine learning is to introduce an additive noise to the data or the update function (e.g., [8, 46]). The parameters of the noise are usually predetermined by the dimensions of the data, the parameters of the machine learning algorithm and the security requirement, and hence are data-independent. As a result, in principle, these constructions can be combined with our system given that the servers can always generate the noise according to the public parameters and add it directly onto the shared values in the training. In this way, the trained model will be differentially private once reconstructed, while all the data still remains private during the training.

Chase et al. [21] consider training neural networks using a hybrid of secure computation and differential privacy. Their technique allows for almost all of the computation to be performed locally by the parties and can, therefore, be significantly more efficient than all previous methods, especially for deep networks. This performance improvement is achieved by updating a public model via a differentially private release of information. In particular, a differentially private gradient of the current model is repeatedly revealed to the participating parties. This approach is also limited to the case where the training data is horizontally partitioned.

3 Preliminaries

3.1 Notation

3.2 Three-party Secure Computation

3.2.1 Secret Sharing Based

Throughout our presentation the default representation of encrypted data uses the replicated secret sharing technique of Araki, et al. [10]. A secret value $x \in \mathbb{Z}_{2^k}$ is shared by sampling three random values $x_1, x_2, x_3 \in \mathbb{Z}_{2^k}$ such that $x = x_1 + x_2 + x_3$. These shares are distributed as the pairs $\{(x_1, x_2), (x_2, x_3), (x_3, x_1)\}$, where party i holds the ith pair. Such a sharing will be denoted as $[\![x]\!]^{\mathsf{A}}$. Sometimes, for brevity, we refer to shares of $[\![x]\!]^{\mathsf{A}}$ as the tuple (x_1, x_2, x_3), though we still mean the replicated secret sharing where each party holds a pair of shares.

First, observe that any two out of the three parties have sufficient information to reconstruct the actual value x. This immediately implies that such a secret sharing scheme can tolerate up to a single corruption. All of the protocols presented will achieve the same threshold. We briefly review these building blocks here. See Table 2 for the cost of each building block. To *reveal* a secret shared value to all parties, party i sends x_i to party $i+1$, and each party reconstructs x locally by adding the three shares. To reveal the secret shared value only to a party i, party $i-1$ sends x_{i-1} to party i who reconstructs the secret locally.

Arithmetic operations can now be applied to these shares. To add two values $[\![x]\!] + [\![y]\!]$ all parties locally compute $[\![x]\!] + [\![y]\!] = [\![x+y]\!] := (x_1 + y_1, x_2 + y_2, x_3 + y_3)$. Addition or subtraction of a public constant with a shared value $[\![x]\!] \pm c = [\![x \pm c]\!]$ can also be done by defining the three shares of $[\![x \pm c]\!]$ as $(x_1 \pm c, x_2, x_3)$. To multiply a shared value $[\![x]\!]$ with a public constant c we can define the shares of $[\![cx]\!]$ as (cx_1, cx_2, cx_3). Note that all of these operations are with respect to the group $\mathbb{Z}_{2^k}$. To multiply two shared values $[\![x]\!]$ and $[\![y]\!]$ together, the parties must interactively compute $[\![xy]\!]$. First observe that,

$$\begin{aligned} xy =& (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ =& x_1y_1 + x_1y_2 + x_1y_3 \\ &+ x_2y_1 + x_2y_2 + x_2y_3 \\ &+ x_3y_1 + x_3y_2 + x_3y_3 \end{aligned}$$

Collectively the parties can compute all such cross terms. We define $[\![z]\!] = [\![xy]\!]$ such that

$$\begin{aligned} z_1 &:= x_1y_1 + x_1y_2 + x_2y_1 \quad + \alpha_1 \\ z_2 &:= x_2y_2 + x_2y_3 + x_3y_2 \quad + \alpha_2 \\ z_3 &:= x_3y_3 + x_3y_1 + x_1y_3 \quad + \alpha_3 \end{aligned}$$

For now ignore the terms $\alpha_1, \alpha_2, \alpha_3$ and observe that party i can locally compute z_i given its shares of $[\![x]\!]$ and $[\![y]\!]$. However, we require that all parties hold two out of the three shares. To ensure this, the protocol specifies that party i sends z_i to party $i-1$. We call this sending operation *re-sharing*. The additional terms $\alpha_1, \alpha_2, \alpha_3$ are used to randomize the shares of z. We therefore require that they be random elements of $\mathbb{Z}_{2^k}$ subject to $\alpha_1 + \alpha_2 + \alpha_3 = 0$. Each party knows exactly one of the three values. Such a triple is referred to as a *zero sharing* and can be computed without any interaction after a one time setup where

party i obtains random keys k_i and k_{i+1} to a pseudorandom function (PRF) F. Then, the jth zero sharing is obtained when party i lets $\alpha_i = F_{k_i}(j) - F_{k_{i+1}(j)}$. For more details see [10].

The same PRF keys can be used to efficiently share a random value in $\mathbb{Z}_{2^k}$. In particular, to share a random value for the jth time, party i lets $r_i = F_{k_i}(j), r_{i+1} = F_{k_{i+1}}(j)$. Note that (r_1, r_2, r_3) form a proper replicated secret sharing of the random value $r = r_1 + r_2 + r_3$.

If, for example, party 1 wishes to construct a sharing of its private input x, the parties first generate another zero sharing $\alpha_1, \alpha_2, \alpha_3$. The shares of $[\![x]\!]$ are then defined as $(x_1, x_2, x_3) := (\alpha_1 + x, \alpha_2, \alpha_3)$. The sharing of x is completed by having party i send the share x_i to party $i - 1$.

The protocols described above work in the semi-honest setting where the parties follow the protocol as described. However, in the case where one of the parties is malicious, they may send the incorrect value during the multiplication or input sharing phases. The follow-up work of [31] presents highly efficient techniques to prevent such attacks. We, therefore, take for granted that all the operations discussed above can be efficiently performed in the malicious setting as well.

Protocol	Malicious		Semi-honest	
	Comm	Round	Comm	Round
Add	0	0	0	0
Mult	$4k$	1	$11k$	1
ZeroShare	0	0	0	0
Rand	0	0	0	0
RevealAll	3	1	6	1
RevealOne	1	1	2	1
Input	3	1	3	1

Table 2: Round is number of messages sent/received, Comm is number of bits exchanged, ring is $\mathbb{Z}_{2^k}$

3.2.2 Arithmetic vs. Binary sharing

Later we will make use of two different versions of the above protocols. The first will correspond to the case of $k = 64$ or some suitably large value which supports traditional arithmetic operations such as +,-,*. We refer to this as arithmetic sharing using the notation $[\![x]\!]^{\mathsf{A}}$. The latter case will be for $k = 1$ where the binary operations $\oplus, \wedge$ correspond to +,*. The advantage of a binary representation is that it can be more flexible and efficient when computing functions that can not easily be framed in terms of modular addition and multiplication. We refer to this as binary sharing and use the notation $[\![x]\!]^{\mathsf{B}}$.

3.2.3 Yao sharing

Two-party sharing In the two-party setting, Yao's garbled circuit protocol allows a garbler to encode a boolean function into a garbled circuit that is evaluated by a second party, called the evaluator. More concretely, the garbling scheme first assigns two random keys $k^0_{\mathsf{w}}, k^1_{\mathsf{w}}$ to each wire w in the circuit corresponding to values 0 and 1 for that wire. Each gate in the circuit is then garbled by encrypting each output wire key using different combinations (according to the truth table for that gate) of input wire keys as encryption keys. These ciphertexts are randomly permuted so their position does not leak real values of the intermediate wires during the evaluation. The evaluator obtains the keys corresponding to input wires to the circuit which enables him to decrypt exactly one ciphertext in each gabled gate and learn the

corresponding output wire key. The evaluator can decode the final output give a translation table that maps the circuit's final output wire keys to their real values.

Various optimizations to this basic garbling idea have been introduced over the years, the most notable of which are the point-and-permute [12], Free-XOR [39] and the half-gate [64] techniques. These optimizations require some modifications to how the keys are generated. In particular, the free-XOR techniques requires that $k^1_{\mathsf{w}} = k^0_{\mathsf{w}} \oplus \Delta$ for every wire w where Δ is a global random (secret) string. To use the point-and-permute technique, we need to let the least significant bit of Δ to be 1, i.e. $\Delta[0] = 1$. The least significant bit of each key ($p_{\mathsf{w}} \oplus i = k^i_{\mathsf{w}}[0]$) is then referred to as the *permutation bit*. As discussed in the two-party ABY framework [26], two-party Yao's sharing of an input bit x for wire w, can be seen as one party holding k^0_{w} and Δ, while the other party holds k^x_{w}.

Three-party sharing Mohassel et al. [47], extend Yao's garbled circuit protocol to the three-party setting with one corruption, and obtain security against a malicious adversary with the cost comparable to that of the semi-honest two-party Yao's protocol. The high-level idea is as follows. Party 1 plays the role of evaluator and parties 2,3 play the role of garblers. The two garblers exchange a random seed that is used to generate all the randomness and keys required by the garbled circuit. They separately generate the garbled circuit and send their copy to the evaluator. Since at least one garbler is honest, one of the garbled circuits is computed honestly. The evaluator can enforce honest garbling behavior by checking equality of the garbled circuits.

The Yao sharing in the three-party setting, denoted by $[\![x]\!]^{\mathsf{Y}}$, can be seen as the evaluator holding k^x_{w} and the two garblers holding $(k^0_{\mathsf{w}}, \Delta)$. In the semi-honest case, a garbler shares its input bit x by sending $k^0_{\mathsf{w}} \oplus x\Delta$ to the evaluator. In the malicious case, both garblers send commitments to both keys (permuted), i.e. $\mathsf{Comm}(k^b_{\mathsf{w}}), \mathsf{Comm}(k^{\neg b}_{\mathsf{w}})$ to the evaluator and the garbler who is sharing its input sends the opening for one of the commitments. The evaluator checks that the two pairs of commitments are equal (the same randomness is used to generate and permute them) and that the opening succeeds. The evaluator could share its input by performing an oblivious transfer with one of the garblers to obtain one of the two keys. Mohassel et al. remove the need for OT by augmenting the circuit such that each input wire corresponding to evaluator is split into two inputs bits that XOR share the original input. The circuit first XORs these two bits (for free) and then computes the expected function. The evaluator shares its input bit x by generating two random bits x_2 and x_3 where $x = x_2 \oplus x_3$ and sending x_i to party i. The party i then shares x_i as it would share its own input, except that there is no need to permute the commitments since party 1 knows the x_is.

To XOR, AND or compute arbitrary boolean functions on the shared values, we just construct the garbled circuit for the corresponding function and proceed as discussed above. All the two-party garbling optimizations such as free-XOR, half-gates and point-and-permute can be used in the three-party setting as well. Note that it is possible to not provide the evaluator with the translation table for output keys such that it only learns the output key but not its real value (hence remain shared). Moreover, it is possible to have the evaluator reveal output values to one or both garbler by sending them the output key. Note that due to the authenticity of the garbling scheme, the evaluator cannot cheat and send the wrong output key.

To share a random value, Parties 2 and 3 generate random values r_2 and r_3 and input them to a garbled circuit that XORs them and lets the evaluator (party 1) learn the output keys. The shared random value is $r = r_2 \oplus r_3$.

Parameters: Clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ and servers $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$.
Uploading Data: On input x_i from $\mathcal{C}_i$, store x_i internally.
Computation: On input f from $\mathcal{S}_1$, $\mathcal{S}_2$, or $\mathcal{S}_3$ compute $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ and send y_i to $\mathcal{C}_i$. This step can be repeated multiple times with different functions.

Figure 1: Ideal Functionality $\mathcal{F}_{ml}$

4 Security Model

We use the same security model and architecture as SecureML [48] except that we extend it to the three party case with an honest majority and consider both semi-honest and malicious adversaries. In particular, data owners (clients) secret share their data among three servers who perform 3PC to train and evaluate models on the joint data. We observe that security in this model reduces to standard security definitions for 3PC between the three servers. Hence, we follow the same security definitions and refer to [10] and [31] for a formal description of these adversarial settings. Since all our building blocks are reduced to the composition of existing 3PC building blocks, their security is implied via standard composition theorems [17].

5 Our Framework

In this section, we construct efficient three-party protocols that form the building blocks of our protocols for training linear and logistic regression, and neural network models. We also provide a general framework for performing mixed computation on shared data, i.e. an equivalent of the ABY framework [26] for the three-party setting. In Section 5.1 we explore protocols for three-party fixed-point multiplication. Somewhat surprisingly, we show that the two-party techniques of [48] do not work in the three-party setting and describe two efficient solutions for this problem that yield approximate fixed-point multiplication on shared values for roughly the same cost as standard integer multiplication. In Section 5.2 we show how decimal arithmetics can be vectorized such that computing the inner product of two n element vectors can be computed with $O(1)$ communication, as opposed to $O(n)$. These two techniques are sufficient to efficiently implement linear regression via the gradient descent algorithm (see Section 6.1). In Section 5.3, we show new and optimized three-party conversion protocols for all possible conversions between arithmetic, binary and Yao sharing of secret values. We then go on to describe additional building blocks that will facilitate the training of logistic regression and neural network models in Section 5.4 and Section 5.5.

5.1 Fixed-point Arithmetic

We now detail how to perform decimal arithmetics by secret sharing fixed-point values.

A fixed point value is defined as a k bit integer using twos-complement representation where the bottom d bits denote the decimal, i.e. for positive values bit i denotes the $(i-d)$th power of 2. Addition and subtraction can be performed using the corresponding integer operation since the results are expected to remain below 2^k. Multiplication can also be performed in the same manner but the number of decimal bits doubles and hence must be divided by 2^d to maintain the d decimal bit invariant.

5.1.1 Why technique of [48] fails

We start by reviewing the two-party fixed-point multiplication protocol of [48] and show why it fails in the three-party setting. [48] secret shares a fixed-point x using the ring $\mathbb{Z}_{2^k}$ as $[\![x]\!] := (x + r, -r)$ for some secret $r \leftarrow \mathbb{Z}_{2^k}$. Addition and subtraction in $\mathbb{Z}_{2^k}$ naturally work but multiplication does not due to (two's complement) division by 2^d not being supported in $\mathbb{Z}_{2^k}$. Consider having a standard sharing $[\![x']\!] := [\![y]\!][\![z]\!]$ over $\mathbb{Z}_{2^k}$ and desire to compute $[\![x]\!] := [\![x'/2^d]\!]$ such that when x, y, z are interpreted as fixed-point values the quality $x = yz$ holds (assuming the semantic values do not overflow) . Ideally both shares of $[\![x']\!] = (x' + r', -r')$ can be locally divided by 2^d to obtain two k-bit shares $[\![\tilde{x}]\!] := (\frac{x'_1}{2^d}, \frac{x'_2}{2^d})$ holding the value $x = x'/2^d = \tilde{x}$. However, this final equality does not hold. First, there is a bad event that the divisions by 2^d removes a carry bit from the first d bits that would have propagated into the $d + 1$th bit. That is, at bit position d of the addition $x'_1 + x'_2 = (x' + r') + (-r') \mod 2^k$ a carry is generated (which we have eliminated due to separately dividing each share by 2^d). However, this probabilistic error has a magnitude of 2^{-d} and is arguably acceptable given that fixed-point arithmetics naturally has limited precision. In fact, [48] shows that this small error, does not have any impact on the accuracy of trained models when d is sufficiently large.

Unfortunately, a more serious error can also be introduced due to the values being shared in the ring modulo 2^k combined with the use of twos complement semantics. In particular, the desired computation of $x'/2^d$ is with respect to two's complement arithmetics, i.e. shift the bits of x' down by d positions and fill the top d bits with the most significant bit (MSB) of x'. This latter step can fail when x' is secret shared in $\mathbb{Z}_{2^k}$. Take for example $x' = -2^{k-1}$, which is represented in binary two's complement as 100...000. We then have that $x'/2^d$ is represented as 1...100...000 where there are $d + 1$ leading ones. However, when secret shared, it is likely that both shares x'_1, x'_2 have zero in the MSB. As a result, when they are divided by 2^d, the two shares will have at least $d + 1$ leading zeros. When these shares are reconstructed the result will be incorrect. In particular, the result will have d (or $d - 1$) leading zeros and correspond to an incorrect *positive* value.

A simple case analysis shows that a necessary condition for this error is that the MSB of x' is opposite of both x'_1, x'_2. That is, the reverse of the example above can also result in this large error[1]. A clever fix to this problem is to maintain that $|x'| < 2^\ell \ll 2^k$ where x' is interpreted as a two's complement integer. This ensures that there is a negligible probability that the MSB of x'_1 is the same as x'_2. To see this, observe that $x'_1 := x' + r', x'_2 = -r'$ and that when $r' \neq 0$ the sign/MSB of r' and $-r'$ are always opposite. When x' is positive the probability of x'_1 having the same MSB as x'_2 is the probability that the top $k - \ell$ bits of r' are all ones and that a carry is generated at the ℓth bit of $x' + r'$. Due to r' being uniformly distributed, the probability that r' has this many leading ones is $2^{\ell-k}$ which can be made very small for appropriately chosen ℓ, k. A similar argument also holds when x' is negative.

In summary, there are two sources of error when performing fixed-point arithmetics. When shares are individually truncated, the carry-in bit can be lost. However, due to this representing an error of magnitude 2^{-d}, the resulting value is "close enough" to be sufficient in most applications. Indeed, the size of this error can be made arbitrarily small simply by increasing the size of d. The other more serious source

[1] In the reversed case, x'_1, x'_2 both have MSB of one which overflows and is eliminated. However, after being sign extended/divided by 2^d, the carry results in $1 + 1 + 1$ in all higher positions, resulting in the d most significant bits being incorrectly set to one since by assumption the MSB of x' is zero.

of error, which has very large magnitude, occurs when the individual shares are sign-extended incorrectly with respect to the underlying value. However, the probability of this event can be made very small.

Unfortunately the approach of truncating each share separately does not extended to three-party secret sharing where $[\![x]\!] = (x + r_1 + r_2, (-r_1), (-r_2))$. The first source of error now has magnitude 2^{-d+1} due to the possibility of truncating two carry bits. However, a more serious issue is that bounding $|x| < 2^\ell$ no longer ensures that the large error happens with very small probability. The necessary condition for this error is more complex due to the possibility of two carry bits, but intuitively, bounding $|x| < 2^\ell$ no longer ensures that exactly one of the shares x_1, x_2, x_3 will be correctly sign-extended due to r_1, r_2 both being uniformly distributed and independent. To make such a secret sharing work, a constraint on how r_1, and r_2 are sampled would be required. However, such a constraint would render the multiplication protocol insecure.

5.1.2 Our Multi-Party Fixed-Point Multiplication

We present two new methods for performing three-party decimal multiplication/truncation with different trade-offs. While presented in terms of three parties, we note that our second technique can be extended to settings with more than three parties as well.

Share Truncation Π_{trunc1} Our first approach minimizes the overall communication at the expense of performing multiplication and truncation in two rounds. The idea is to run the two-party protocol where one party does not participate. Since we assume an honest majority, the security still holds in the semi-honest setting. Let the parties hold a 2-out-of-3 sharing of $[\![x']\!] := [\![y]\!][\![z]\!]$ over the ring $\mathbb{Z}_{2^k}$ and desire to compute $[\![x]\!] = [\![x']\!]/2^d$. As in the two-party case, we assume that $x' \ll 2^k$.

Parties begin by defining the 2-out-of-2 $(x'_1, x'_2 + x'_3)$ between party 1 and 2, and locally truncate their shares to $(x'_1/2^d, (x'_2 + x'_3)/2^d)$. The errors introduced by the division mirror that of the two-party case and guarantees the same correctness. The result is defined as $[\![x]\!] := (x_1, x_2, x_3) = (x'_1/2^d, (x'_2 + x'_3)/2^d - r, r)$, where $r \in \mathbb{Z}_2^k$ is a random value known to parties 2,3. Note that party i can locally compute the share x_i and therefore $[\![x]\!]$ can be made a 2-out-of-3 sharing by sending x_i to party $i - 1$. One limitation of this approach is that two rounds are required to multiply and truncate. Crucially, party 2 must know both x'_2 and x'_3 where the latter is computed by party 3 in the first round of multiplication where the parties compute a 3-out-of-3 sharing of $[\![x']\!] := [\![y]\!][\![z]\!]$. Then party 3 sends x'_3 to party 2 who then computes $x_2 := (x'_2 + x'_3)/2^d - r$ and forwards this value to party 1. A complete description of the protocol appears in Figure 2.

Share Truncation Π_{trunc2} The number of multiplication rounds can be reduced back to 1 with a more sophisticated technique which leverages preprocessing. First, let us assume we have preprocessed the shares $[\![r']\!], [\![r]\!] = [\![r'/2^d]\!]$ where $r' \in \mathbb{Z}_{2^k}$ is random. Again, let us have computed $[\![x']\!]$ over the ring $\mathbb{Z}_{2^k}$ and wish to divide it by 2^d. To compute the sharing of $x = yz/2^d$ we first reveal $[\![x' - r']\!] = [\![x']\!] - [\![r']\!]$ to all parties[2]. Locally, everyone can compute $(x' - r')/2^d$. Parties then collectively compute $[\![\widetilde{x}]\!] := (x' - r')/2^d + [\![r]\!]$. Note that this computation exactly emulates the two-party scenario and therefore the maximum error between

[2]Revealing to two parties is sufficient.

Parameters: A single 2-out-of-3 share $[\![x']\!]^{\mathsf{A}} = (x'_1, x'_2, x'_3)$ over $\mathbb{Z}_{2^k}$, a positive integer d, and a pre-shared key K between party 2,3 for a PRF F.

1. Parties 2,3 locally compute a random $r \in \mathbb{Z}_{2^k}$ by invoking $F_K(\cdot)$.
2. Party 1,3 locally compute $x_1 := x'_1/2^d$.
3. Party 2 locally computes $x_2 := (x'_2 + x'_3)/2^d - r$ and sends this to party 1.
4. Party 3,2 locally compute $x_3 := r$.
5. Output $[\![y]\!]^{\mathsf{A}} := (x_1, x_2, x_3)$.

Figure 2: Semi-honest share truncation protocol Π_{trunc1}. Requires two rounds when composed with fixed-point multiplication.

$\widetilde{x}$ and $x = yz/2^d$ will be 2^{-d} with probability $1 - 2^{-k+\ell}$ which is overwhelming for correctly chosen k and ℓ where $x \in \mathbb{Z}_{2^\ell}$.

This operation can be combined with the computation of $[\![x']\!] := [\![y]\!][\![z]\!]$ and performed in a single round. Recall that standard share multiplication is performed in two steps, 1) locally compute a 3-out-of-3 sharing of $[\![x']\!]$, and 2) reshare it as a 2-out-of-3 sharing. Between steps 1 and 2, the parties can instead compute a 3-out-of-3 sharing of $[\![x' - r']\!]$. Step 2) can then be replaced by revealing $[\![x' - r']\!]$ and defining $[\![x]\!] := (x' - r')/2^d + [\![r]\!]$. So the multiplication and truncation can be done in exactly one round and the required communication is 4 messages as opposed to 3 in standard multiplication.

There are several ways to compute the pair $[\![r']\!], [\![r]\!] = [\![r'/2^d]\!]$. The most immediate approach could be to use Π_{Trunc1}, but we choose to use a more communication efficient method using binary secret sharing that is also secure against malicious adversaries. First, parties non-interactively compute the random binary share $[\![r']\!]^{\mathsf{B}}$. This sharing is locally truncated to obtain $[\![r]\!]^{\mathsf{B}}$ by removing the bottom d shares. To obtain the final sharing $[\![r']\!]^{\mathsf{A}}, [\![r]\!]^{\mathsf{A}}$, parties 1 and 2 jointly sample and secret share the values $r'_2, r_2 \in \mathbb{Z}_{2^k}$ and parties 2 and 3 sample and share r'_3, r_3 in the same way (i.e. generating them using pre-shared PRF keys). Parties then securely compute subtraction binary circuits, and reveal $[\![r'_1]\!]^{\mathsf{B}} := [\![r']\!]^{\mathsf{B}} - [\![r'_2]\!]^{\mathsf{B}} - [\![r'_3]\!]^{\mathsf{B}}$ and $[\![r_1]\!]^{\mathsf{B}} := [\![r]\!]^{\mathsf{B}} - [\![r_2]\!]^{\mathsf{B}} - [\![r_3]\!]^{\mathsf{B}}$ to party 1 and 3. The final shares are defined as $[\![r']\!] := (r'_1, r'_2, r'_3)$ and $[\![r]\!] := (r_1, r_2, r_3)$.

This computation can be performed in parallel for all truncations in a preprocessing stage and hence has little impact on the overall round complexity of the protocol. As a result, we choose to optimize the overall communication (instead of rounds) of the addition circuit with the use of an optimized ripple carry full addition/subtraction circuit using $k - 1$ AND gates. As an additional optimization, the computation of $[\![r_1]\!]$ can be performed in $\mathbb{Z}_{2^{k-d}}$ and therefore requires $k - d - 1$ AND gates per subtraction. In the semi-honest setting, one of the subtractions of r_2, r_3 can be performed locally by party 2.

Another advantage of this second protocol is its compatibility with the malicious setting. When the computation of $[\![x']\!] = [\![y]\!][\![z]\!]$ is performed initially all parties hold a 3-out-of-3 sharing of $[\![x']\!]$ and then reshare this to be a 2-out-of-3 sharing by sending x'_i to party $i - 1$. Additionally, a small proof π_i is sent demonstrating that x'_i is indeed the correct value. We propose that this x'_i and the proof is still sent along with the reveal of $[\![x' - r']\!]$ which can be composed into a single round. However, it is possible for party i to send the correct message (x_i, π_i) to party $i - 1$ but send the incorrect reveal message $x_i - r_i$ to party $i + 1$. To ensure that such behavior is caught, parties $i - 1$ and $i + 1$ should maintain a transcript of all $x_i - r_i$ messages from party i and compare them for equality before any secret value is revealed. For a more detailed description of the protocol and a security analysis, we refer the reader to Section 8.1.

Parameters: A single 2-out-of-3 (or 3-out-of-3) share $[\![x']\!]^{\mathsf{A}} = (x'_1, x'_2, x'_3)$ over the ring $\mathbb{Z}_{2^k}$ and a integer $d < k$.
Preprocess:

1. All parties locally compute $[\![r']\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^k)$.
2. Define the sharing $[\![r]\!]^{\mathsf{B}}$ to be the $k-d$ most significant shares of $[\![r']\!]^{\mathsf{B}}$, i.e. $r = r'/2^d$.
3. The parties compute $[\![r'_2]\!]^{\mathsf{B}}, [\![r'_3]\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^k)$ and $[\![r_2]\!]^{\mathsf{B}}, [\![r_3]\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^{k-d})$. r'_2, r_2 is revealed to party 1,2 and r'_3, r_3 to parties 2,3 using the $\mathsf{RevealOne}$ routine.
4. Using a ripple carry subtraction circuit, the parties jointly compute $[\![r'_1]\!]^{\mathsf{B}} := [\![r']\!]^{\mathsf{B}} - [\![r'_2]\!]^{\mathsf{B}} - [\![r'_3]\!]^{\mathsf{B}}$, $[\![r_1]\!]^{\mathsf{B}} := [\![r]\!]^{\mathsf{B}} - [\![r_2]\!]^{\mathsf{B}} - [\![r_3]\!]^{\mathsf{B}}$ and reveal r'_1, r_1 to parties 1,3.
5. Define the preprocessed shares as $[\![r']\!]^{\mathsf{A}} := (r'_1, r'_2, r'_3), [\![r]\!]^{\mathsf{A}} := (r_1, r_2, r_3)$.

Online:

1. The parties jointly compute $[\![x' - r']\!]^{\mathsf{A}}$ and then compute $(x' - r') := \mathsf{RevealAll}([\![x - r']\!]^{\mathsf{A}})$.
2. Output $[\![x]\!]^{\mathsf{A}} := [\![r]\!]^{\mathsf{A}} + (x' - r')/2^d$.

Figure 3: Single round share truncation protocol Π_{trunc2}.

Public Operations One useful property of an additively secret shared value $[\![x]\!]^{\mathsf{A}}$ is that $c + [\![x]\!]^{\mathsf{A}}, [\![x]\!]^{\mathsf{A}} - c, c[\![x]\!]^{\mathsf{A}}$ for any signed integer c can be computed locally. When x is a fixed-point value, addition and subtraction naturally work so long as c is also expressed as a fixed-point value. For multiplication and a two's complement integer c, the standard multiplication with a public value still works. When c is a fixed point value, the result must be divided by 2^d using the semi-honest Π_{trunc1} or malicious Π_{trunc2} protocol to obtain a sharing $[\![cx]\!]^{\mathsf{A}}$ with d decimal bits. One byproduct of fixed-point multiplication is that division by a public value c is now supported very efficiently , i.e. $[\![x]\!]^{\mathsf{A}}/c = c^{-1}[\![x]\!]^{\mathsf{A}}$.

5.2 Vectorized Multiplication

For many machine learning algorithms the primary computation is matrix multiplication. This in turn can be implemented by a series of inner products, one for each row-column pair of the first and second matrices. Inner product is defined as $\vec{x} \cdot \vec{y} := \sum_{i=1}^{n} x_i y_i$, where $\vec{x}, \vec{y} \in (\mathbb{Z}_{2^k})^n$ are vectors of n elements. A naive solution would require n independent multiplication protocols and $O(n)$ communication. We show how this can be optimized to only require communicating $O(1)$ ring elements, and computing only one pre-processed truncation-pair $[\![r']\!], [\![r]\!]$.

Recall from the previous section that semi-honest decimal multiplication is performed in two steps by first revealing the 3-out-of-3 sharing $[\![z' + r']\!] = [\![x]\!][\![y]\!] + [\![r']\!]$. The final product is then computed as $[\![z]\!] := (z' + r')/2^d - [\![r]\!]$. Observe that the primary non-linear step here is the computation of $[\![x]\!][\![y]\!]$ after which a series of local transformations are made. As such, the computation of the inner product can be written as $[\![\vec{x}]\!] \cdot [\![\vec{y}]\!] := \mathsf{reveal}((\sum_{i=1}^{n} [\![x_i]\!][\![y_i]\!]) + [\![r']\!])/2^d - [\![r]\!]$. Here, all parties locally compute a 3-out-of-3 sharing of each $[\![x_i]\!][\![y_i]\!]$ which are summed, masked, truncated, and reshared as a 2-out-of-3 sharing of the final result. As a result, only a single value is reshared. One additional benefit of this approach is that the truncation induces an error of 2^{-d} with respect to the overall inner produce, as opposed to individual multiplication terms, resulting in a more accurate computation. More generally, any linear combination of multiplication terms can be computed in this way where the parties communicate to reshare and truncate only after computing the 3-out-of-3 secret share of the linear combination (as long as the final result does not grow beyond the 2^ℓ bound). The malicious setting is more complicated due o the fact that for each multiplication $[\![x_i]\!][\![y_i]\!]$ a proof of correctness must be provided. This would

immediately result in the communication increasing back to $O(n)$ elements. However, we show that in the context of matrix multiplication this increased communication can be transferred to an offline phase. To compute $[\![X]\!][\![Y]\!]$ the parties first generate two random matrices $[\![A]\!], [\![B]\!]$ which are respectively the same dimension as $[\![X]\!], [\![Y]\!]$. During the offline phase, the parties compute the matrix triple $[\![C]\!] := [\![A]\!][\![B]\!]$ using the scalar fixed-point multiplication protocol described in the previous section. Given this, the malicious secure multiplication protocol of [31] can naturally be generalized to the matrix setting. In particular, the parties locally compute the 3-out-of-3 sharing $[\![Z]\!] := [\![X]\!][\![Y]\!]$ and then party i sends their local share Z_i to party $i-1$. Party i also proves the correctness of Z_i using the matrix triple $([\![A]\!], [\![B]\!], [\![C]\!])$ along with a natural extension of protocol 2.24 in [31] where scaler operations are replaced with matrix operations. The online communication of this protocol is proportional to the sizes of X, Y, Z and almost equivalent to the semi-honest protocol. However, the offline communication is proportional to the number of scaler multiplication which is cubic in the dimensions of X and Y.

5.3 Share Conversions

For many machine learning functions, it is more efficient to switch back an forth between arithmetic (multiplications and addition) and binary (non-linear activation functions, max-pooling, averages, etc.) operations. In such cases, it is necessary to convert between different share representations. We design new and optimized protocols that facilitate efficient conversions between all three types of sharing: arithmetic, binary and Yao. We elaborate on these next. See Table 1 for the cost of various conversions.

Bit Decomposition, $[\![x]\!]^{\mathsf{A}} \to [\![\vec{x}]\!]^{\mathsf{B}}$ We begin with bit decomposition where an arithmetic sharing of $x \in \mathbb{Z}_{2^k}$ is converted to a vector of secret shared bits $x[1], ..., x[k] \in \{0,1\}$ such that $x = \sum_{i=1}^{k} 2^{i-1} x[i]$. The basic idea is that parties use their shares of $[\![x]\!]^{\mathsf{A}} = (x_1, x_2, x_3)$ as input to a boolean circuit that computes their sum. But we introduce several optimizations that significantly reduce rounds of communication and the communication complexity of this approach. Observe that $[\![x]\!]^{\mathsf{A}} = (x_1, x_2, x_3)$ can be converted to $[\![x_1]\!]^{\mathsf{B}} := (x_1, 0, 0), [\![x_2]\!]^{\mathsf{B}} := (0, x_2, 0), [\![x_3]\!]^{\mathsf{B}} := (0, 0, x_3)$ with no communication. The parties then compute $[\![x]\!]^{\mathsf{B}} = [\![x_1]\!]^{\mathsf{B}} + [\![x_2]\!]^{\mathsf{B}} + [\![x_3]\!]^{\mathsf{B}}$. In general this deterministic sharing can be insecure due to the possibility of leaking information through taking linear combinations of these terms. For example, revealing $[\![x_1]\!]^{\mathsf{B}} \oplus [\![x_2]\!]^{\mathsf{B}}$ leaks both x_1 and x_2 as opposed to the sum of the two shares. However, in our case this does not cause a problem due to how x_1, x_2, x_3 are distributed between the parties. Take the example above, where all parties can recover x_1, x_2 given $x_1 \oplus x_2$ since they already hold at least one of the shares. Naively using the textbook ripple-carry full adder (RCFA) circuit would require $2k$ rounds to compute $\mathsf{RCFA}(\mathsf{RCFA}(x_1, x_2), x_3)$ when performing 3PC on binary shared values. To avoid the high round complexity which becomes the bottleneck in our implementations, we first employ a parallel prefix adder (PPA) [36] which takes two inputs and computes the sum using a divide and conquer strategy, totaling $\log k$ rounds and $k \log k$ gates. Once again, doing this naively would require two addition circuits. We show how to keep the cost close to that of a single PPA in both the semi-honest and the malicious setting, hence reducing both the round (only for binary sharing) and communication complexity by a factor of two.

First observe that the computation of $x_1 + x_2 + x_3$ can be reduced to computing $2c + s$ by executing k independent full adders $\mathsf{FA}(x_1[i], x_2[i], x_3[i-1]) \to (c[i], s[i])$ for $i \in \{0, ..., k-1\}$ where $c[i], s[i]$ denote the ith bit of the bitstrings c and s. It is worth noting that traditionally, full adders are chained together to compute the addition of two bits and a carry in, however, here we used them to reduce 3 operands (i.e.

$x_1, x_2, x_3)$ to 2 (i.e. c, s) while using a single round of communication as opposed to k. The final result can then be computed as $2[\![c]\!]^{\mathsf{B}} + [\![s]\!]^{\mathsf{B}}$ using a parallel prefix adder. Alternatively, in the semi-honest setting Party 2 can provide $(x_1 + x_2)$ as *private* input to a 3PC which computes $[\![x]\!]^{\mathsf{B}} := [\![x_1 + x_2]\!]^{\mathsf{B}} + [\![x_3]\!]^{\mathsf{B}}$. In both settings, this results in a total of $1 + \log k$ rounds, which is significantly better than a factor of two increase in rounds and communication.

Bit Extraction, $[\![x]\!]^{\mathsf{A}} \to [\![x[i]]\!]^{\mathsf{B}}$ A special case of bit decomposition is when a single bit of the share $[\![x]\!]^{\mathsf{A}}$ should be decomposed into a binary sharing, e.g. the ith bit $[\![x[i]]\!]^{\mathsf{B}}$. This case can be simplified such that only $O(i)$ AND gates and $O(\log i)$ rounds are required. While relatively simple, this optimization removes all the unnecessary gates from the parallel prefix adder. As a result, the circuit logic only requires $2i$ AND gates. We use this optimization in our implementation. For brevity, we refer the reader to inspect [36] to deduce exactly which gates can be removed.

Bit Composition, $[\![x]\!]^{\mathsf{B}} \to [\![x]\!]^{\mathsf{A}}$ It can also be required to convert a k bit value in the binary secret share representation to an arithmetic secret share representation. Effectively, we use the same circuit as the bit decomposition with the order of operations slightly reversed. First, parties 1,2 input a random share $[\![-x_2]\!]^{\mathsf{B}}$ and parties 2,3 input a random share $[\![-x_3]\!]^{\mathsf{B}}$. These will be part of the final arithmetic sharing and therefore the former can be known to parties 1,2 and the latter can be known to parties 2,3. $[\![x_2]\!]^{\mathsf{B}}$ can be generated by having parties 1,2 hold three PRF keys $\kappa_1, \kappa_2, \kappa_3$ and party 3 hold κ_2, κ_3. The share is then defined as $[\![x_2]\!]^{\mathsf{B}} := (F(\kappa_1, N), F(\kappa_2, N), F(\kappa_3, N))$ where N denotes a public nonce. $[\![x_3]\!]^{\mathsf{B}}$ can be defined in a similar way with the roles shifted.

The parties compute $\mathsf{FA}([\![x[i]]\!]^{\mathsf{B}}, [\![-x_2[i]]\!]^{\mathsf{B}}, [\![-x_3[i]]\!]^{\mathsf{B}}) \to ([\![c[i]]\!]^{\mathsf{B}}, [\![s[i]]\!]^{\mathsf{B}})$ for $i \in \{1, ..., k-1\}$ and then using a parallel prefix adder $[\![x_1]\!]^{\mathsf{B}} := 2[\![c]\!]^{\mathsf{B}} + [\![s]\!]^{\mathsf{B}}$. In the semi-honest setting, this can be further optimized by having party 2 provide $(-x_2 - x_3)$ as private input and compute $[\![x_1]\!]^{\mathsf{B}} := [\![x]\!]^{\mathsf{B}} + [\![-x_2 - x_3]\!]^{\mathsf{B}}$ using a parallel prefix adder. Regardless, x_1 is revealed to parties 1,3 and the final sharing is defined as $[\![x]\!]^{\mathsf{A}} := (x_1, x_2, x_3)$. Overall, the conversion requires $1 + \log k$ rounds and $k + k \log k$ gates.

Bit Injection, $[\![x]\!]^{\mathsf{B}} \to [\![x]\!]^{\mathsf{A}}$ Another special case can often occur when a single bit x encoded in a binary sharing needs to be promoted to an arithmetic sharing $[\![x]\!]^{\mathsf{A}}$. For ease of presentation, we defer the explanation of this technique to Section 5.4 where a generalization of it to efficiently compute $a[\![x]\!]^{\mathsf{B}} \to [\![ax]\!]^{\mathsf{A}}$ is presented.

Joint Yao Input Recall that in Yao sharing of a bit x, party 1 (evaluator) holds k_{x}^x while the other two parties hold $k_{\mathsf{x}}^0 \in \{0,1\}^\kappa$ and a global random $\Delta \in \{0,1\}^\kappa$ such that $k_{\mathsf{x}}^1 := k_{\mathsf{x}}^0 \oplus \Delta$. A useful primitive for conversions to and from Yao shares is the ability for two parties to provide an input that is known to both of them. For example, parties 1,2 know a bit x and wish to generate a sharing of $[\![x]\!]^{\mathsf{Y}}$. In the semi-honest setting, this is trivial as party 2 can locally generate and send $[\![x]\!]^{\mathsf{Y}}$ to party 1 (who uses it to evaluate a garbled circuit). However, in the malicious setting party 1 needs to verify that $[\![x]\!]^{\mathsf{Y}}$ actually encodes x without learning Δ. In the current example, party 3 can be used to allow party 1 to check the correctness of the sharing by having party 1 and 3 send $\mathsf{Comm}(k_{\mathsf{x}}^0), \mathsf{Comm}(k_{\mathsf{x}}^1)$ generated using the same randomness shared between them (party 2 can send a hash of the commitments). Party 1 verifies that both parties sent the same commitments and that $\mathsf{Comm}(k_{\mathsf{x}}^x)$ decommits to k_{x}^x. This interaction requires two commitments,

one decommitment and at most one round per input bit. In the case that x is known to parties 1,3 the roles of 2,3 above can simply be reversed.

When sharing many input bits ($n \gg \lambda$, for λ a statistical security parameter), we show that the number of commitments can be capped at 2λ. After receiving the input labels $k^{x_1}_{\mathsf{x}_1}, ..., k^{x_n}_{\mathsf{x}_n}$ (without commitments) and before revealing any secret values which are dependent on these input labels, party 1 computes λ random linear combinations $k^{c_1}_{\mathsf{c}_1}, ..., k^{c_\lambda}_{\mathsf{c}_\lambda}$ of $k^{x_1}_{\mathsf{x}_1}, ..., k^{x_n}_{\mathsf{x}_n}$ in $(\mathbb{Z}_2)^\lambda$ with coefficients in $\mathbb{Z}_2$. Parties 2, 3 receive the combination from party 1 and both compute the λ combinations of $k^0_{\mathsf{x}_1}, ..., k^0_{\mathsf{x}_n}$ to obtain $k^0_{\mathsf{c}_1}, ..., k^0_{\mathsf{c}_\lambda}$. Using the same randomness, parties 2,3 send $\mathsf{Comm}(k^0_{\mathsf{c}_i}), \mathsf{Comm}(k^1_{\mathsf{c}_i} = k^0_{\mathsf{c}_i} \oplus \Delta)$ for $i \in \{1, ..., \lambda\}$ to party 1 (one party can send hash of the commitments instead). Party 1 verifies that the two sets of commitment are the same and that $\mathsf{Comm}(k^{c_i}_{\mathsf{c}_i})$ decommits to $k^{c_i}_{\mathsf{c}_i}$ for all i. The probability that party 1 received an incorrect label and this test passes is $2^{-\lambda}$. In particular, consider that one of the garblers sends an incorrect input label. To have the ith linear combination pass, either this input label must not be in the sum (happens with Pr. 1/2) or was canceled out by another incorrect label ℓ (canceling out using multiple incorrect labels only reduces adversaries winning probability). Fixing all previous labels, the probability that ℓ is included in the sum is 1/2. As a result, with either strategy (including additionl incorrect labels, or not), the probability of undetected cheating in one linear combination is bounded by 1/2. We therefore have that cheating is caught with probability $1 - 2^{-\lambda}$ and set λ to be the statistical security parameter to ensure that cheating is undetected with only a negligible probability.

In the other case were 2,3 both know x, it is possible to generate $[\![x]\!]^{\mathsf{Y}}$ with no communication. Using a shared (among all three parties) source of randomness, the parties locally sample $k^x_{\mathsf{x}} \leftarrow \{0,1\}^\kappa$. Parties 2,3 can then define $k^0_{\mathsf{x}} := k^x_{\mathsf{x}} \oplus (x\Delta)$.

Yao to Binary, $[\![x]\!]^{\mathsf{Y}} \rightarrow [\![x]\!]^{\mathsf{B}}$ As observed in [26], the least significant bit of the keys (permutation bit) form a two-party sharing of x. i.e. $x \oplus p_{\mathsf{x}} = k^x_{\mathsf{x}}[0]$ where $p_{\mathsf{x}} = k^0_{\mathsf{x}}[0]$. Note that party 3 also knows p_{x} since it holds $k^0_{\mathsf{x}}[0]$. Parties 1 and 2 locally generate another random bit r and party 1 sends $k^x_{\mathsf{x}}[0] \oplus r = x \oplus p_{\mathsf{x}} \oplus r$ to Party 3. This yields the following three-party replicated sharing $[\![x]\!]^{\mathsf{B}} = (x \oplus p_{\mathsf{x}} \oplus r, r, p_{\mathsf{x}})$ in a single round and with one bit of communication.

In the malicious setting, the bit $x \oplus b \oplus r$ that party 1 sends to party 3 must be authenticated to ensure that party 1 indeed uses $b = p_{\mathsf{x}}$. Parties 1 and 2 sample $k^r_{\mathsf{r}} \leftarrow \{0,1\}^\kappa$ and party 2 sends $k^0_{\mathsf{r}} := k^r_{\mathsf{r}} \oplus (r\Delta)$ to party 3 who sends commitments of $k^0_{\mathsf{y}} := k^0_{\mathsf{x}} \oplus k^0_{\mathsf{r}}, k^1_{\mathsf{y}} := k^0_{\mathsf{x}} \oplus k^0_{\mathsf{r}} \oplus \Delta$ to party 1. Party 1 sends $k^{x\oplus r}_{\mathsf{y}} := k^x_{\mathsf{x}} \oplus k^r_{\mathsf{r}}$ to party 3 who verifies that it is in the set $\{k^0_{\mathsf{y}}, k^1_{\mathsf{y}}\}$. Party 1 also verifies that the commitment $\mathsf{Comm}(k^{x\oplus r}_{\mathsf{y}})$ sent by party 3 decommits to $k^{x\oplus r}_{\mathsf{y}}$. The parties can then compute the three-party sharing $[\![x]\!]^{\mathsf{B}} = (x \oplus p_{\mathsf{x}} \oplus r, r, p_{\mathsf{x}})$. Observe that party 3 computes $x \oplus p_{\mathsf{x}} \oplus r$ as $k^{x\oplus r}_{\mathsf{y}}[0] \oplus p_{\mathsf{r}}$. In total, this conversion takes two rounds of communication, however, the final sharing $[\![x]\!]^{\mathsf{B}}$ is computable after a single round. It is therefore ok to use $[\![x]\!]^{\mathsf{B}}$ after the first round so long as dependent values are not revealed in that round. In the event that the verification steps fail, the parties should abort.

Binary to Yao, $[\![x]\!]^{\mathsf{B}} \rightarrow [\![x]\!]^{\mathsf{Y}}$ where $[\![x]\!]^{\mathsf{B}} = (x_1, x_2, x_3)$. Parties jointly input the shares $[\![x_1]\!]^{\mathsf{Y}}, [\![x_2]\!]^{\mathsf{Y}}, [\![x_3]\!]^{\mathsf{Y}}$ using the procedure discuss earlier for joint Yao input. The final share can then be computed using a garbled circuit that computes XOR of the three values, i.e. $[\![x]\!]^{\mathsf{Y}} := [\![x_1]\!]^{\mathsf{Y}} \oplus [\![x_2]\!]^{\mathsf{Y}} \oplus [\![x_3]\!]^{\mathsf{Y}}$. With the free-XOR technique, this does not require any communication between the parties and can be computed locally by party 1. In the semi-honest setting, this can be further optimized by observing that party 2 knows x_2

and x_3. They can therefore locally compute $x_2 \oplus x_3$ and send $[\![x_2 \oplus x_3]\!]^{\mathsf{Y}}$ to party 1 who locally computes $[\![x]\!]^{\mathsf{Y}} := [\![x_1]\!]^{\mathsf{Y}} \oplus [\![x_2 \oplus x_3]\!]^{\mathsf{Y}}$.

Yao to Arithmetic, $[\![x]\!]^{\mathsf{Y}} \to [\![x]\!]^{\mathsf{A}}$ To convert $x \in \mathbb{Z}_{2^k}$ from Yao to arithmetic sharing, we could first switch from Yao to Binary and then perform the bit composition or biy injection (in case of a single bit) protocol, but since the inputs are in form of Yao sharings, we choose to use a garbled circuit 3PC for the CRFA addition circuit. Parties 1, 2 sample $x_2 \leftarrow \mathbb{Z}_{2^k}$ and parties 2, 3 sample $x_3 \leftarrow \mathbb{Z}_{2^k}$ and jointly input them using the procedures above. Then, using a garbled circuit parties compute $[\![x_1]\!]^{\mathsf{Y}} := [\![x]\!]^{\mathsf{Y}} - [\![x_2]\!]^{\mathsf{Y}} - [\![x_3]\!]^{\mathsf{Y}}$ and reveal this to parties 1 and 3. $[\![x]\!]^{\mathsf{A}} = (x_1, x_2, x_3)$ forms the new arithmetic sharing of x. This requires communicating k joint input bits (only x_2) and $2k$ garbled gates. In the semi-honest setting this can be further optimized by having party 3 locally compute $x_2 + x_3$ and provide it as private input to $[\![x_1]\!]^{\mathsf{Y}} := [\![x]\!]^{\mathsf{Y}} - [\![x_2 + x_3]\!]^{\mathsf{Y}}$. As a result, the cost of the garbled circuit is reduced by a factor of 2.

Arithmetic to Yao, $[\![x]\!]^{\mathsf{A}} \to [\![x]\!]^{\mathsf{Y}}$ Parties jointly input the shares of $[\![x]\!]^{\mathsf{A}} = (x_1, x_2, x_3)$ to generate $[\![x_1]\!]^{\mathsf{Y}}, [\![x_2]\!]^{\mathsf{Y}}, [\![x_3]\!]^{\mathsf{Y}}$. A garbled circuit can then be used to generate $[\![x]\!]^{\mathsf{Y}} := [\![x_1]\!]^{\mathsf{Y}} + [\![x_2]\!]^{\mathsf{Y}} + [\![x_3]\!]^{\mathsf{Y}}$. In the semi-honest setting this can be optimized by having party 2 locally compute $x_2 + x_3$ and send party 1 the sharing $[\![x_2 + x_3]\!]^{\mathsf{Y}}$ who computes and the final sharing $[\![x]\!]^{\mathsf{Y}} := [\![x_1]\!]^{\mathsf{Y}} + [\![x_2 + x_3]\!]^{\mathsf{Y}}$.

5.4 Computing $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$

While converting between share representations allows for arbitrary combination of shares to be used together, it can be more efficient to provide custom protocols to directly perform the computation on mixed representation. To this end, we provide a mixed-protocol for performing $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$. This operation is needed repeatedly to compute piecewise linear or polynomial functions that are commonly used to approximate non-linear activation functions in training logistic regression and neural network models. All of these operations will take a shared binary bit $[\![b]\!]^{\mathsf{B}}$ and multiply it by an arithmetic (possibly shared) value $a \in \mathbb{Z}_{2^k}$ and output an arithmetic sharing $[\![c]\!]^{\mathsf{A}} := [\![ab]\!]^{\mathsf{A}}$. The difficulty in constructing such protocols is that $b \in \{0, 1\}$ is shared over $\mathbb{Z}_2$, i.e. $b = b_1 \oplus b_2 \oplus b_3$. However, the result needs to be shared over $\mathbb{Z}_{2^k}$ as $c = c_1 + c_2 + c_3 \mod 2^k$.

5.4.1 Semi-honest Security

Three-Party OT We begin by providing an oblivious transfer protocol in the three-party honest majority setting. As with the two-party 1-out-of-2 OT case, we have a sender and a receiver. To this, we add a third party called a *helper* who receives no output and knows the receiver's choice bit. The functionality for the (sender, receiver, helper) can be expressed as $((m_0, m_1), c, c) \mapsto (\perp, m_c, \perp)$. Several previous work consider multi-party OT [49, 30, 19, 40], but to the best of our knowledge we are the first to consider this particular functionality with an honest majority.

Our approach is extremely efficient with information-theoretic security. The sender and helper first sample two random strings $w_0, w_1 \leftarrow \{0, 1\}^k$ known to both of them. The sender masks the messages as $m_0 \oplus w_0, m_1 \oplus w_1$ and sends them to the receiver. The helper knows that the receiver desires the message m_c. As such the helper sends w_c to the receiver who can then recover m_c. This procedure requires sending 3 messages in a single round.

Computing $a[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ The simplest case is the multiplication of a public value $a \in \mathbb{Z}_{2^k}$ known to party 1 with a shared bit $b \in \{0,1\}$. First, party 3 (the sender) samples $r \leftarrow \mathbb{Z}_{2^k}$ and defines two messages, $m_i := (i \oplus b_1 \oplus b_3)a - r$ for $i \in \{0,1\}$. Party 2 (the receiver) defines his input to be b_2 in order to learn the message $m_{b_2} = (b_2 \oplus b_1 \oplus b_3)a - r = ba - r$. Note that party 1 (the helper) also knows the value b_2 and therefore the three party OT protocol above can be used here. The parties then use locally generated replicated zero sharing (s_1, s_2, s_3) to compute $[\![c]\!] = [\![ab]\!] = (s_1 + r, ab - r + s_3, s_3)$. However, to make this a valid 2-out-of-3 secret sharing, $c_2 = ab - r + s_3$ must be sent to party 1. This would result in a total of two rounds of communications. Alternatively, the three-party OT procedure can be repeated (in parallel) with again party 3 playing the sender with inputs $(i + b_2 + b_3)a - r + s_3$ for $i \in \{0,1\}$ so that party 1 (the receiver) with input bit b_2 learns the message c_2 (not m_{b_2}) in the first round, totaling $6k$ bits and 1 round.

Computing $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ In the semi-honest setting, it is sufficient to run the $a[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ procedure twice in parallel. Crucial in this technique is to observe that a in the computation above need not be public. That is, party 1 could have privately chosen the value of a. Leveraging this, observe that the expression can be written as $[\![a]\!][\![b]\!]^{\mathsf{B}} = a_1[\![b]\!]^{\mathsf{B}} + (a_2 + a_3)[\![b]\!]^{\mathsf{B}}$. Party 1 acts as the sender for the first term and party 3 for the second term. In total $4k$ bits per party are communicated over 1 round.

5.4.2 Malicious Security

Computing $a[\![b]\!]^{\mathsf{B}} = [\![ab]\!]^{\mathsf{A}}$ Unfortunately, our semi-honest approach fails in the malicious setting primarily due to party 1 being free to choose the value a it inputs to the OT, arbitrarily. We avoid this issue by first performing *bit injection* on b. That is, we compute $[\![b]\!]^{\mathsf{B}} \to [\![b]\!]^{\mathsf{A}}$ and then $a[\![b]\!]^{\mathsf{A}} = [\![ab]\!]^{\mathsf{A}}$. As performed in Section 5.3, the parties can locally compute shares $[\![b_1]\!]^{\mathsf{A}}, [\![b_2]\!]^{\mathsf{A}}, [\![b_3]\!]^{\mathsf{A}}$ where $[\![b]\!]^{\mathsf{B}} = (b_1, b_2, b_3)$. We can now emulate the XOR of these values within an arithmetic circuit by computing $[\![b_1 \oplus b_2]\!]^{\mathsf{A}} = [\![d]\!]^{\mathsf{A}} := [\![b_1]\!]^{\mathsf{A}} + [\![b_1]\!]^{\mathsf{A}} - 2[\![b_1]\!]^{\mathsf{A}}[\![b_2]\!]^{\mathsf{A}}$ followed by $[\![b]\!]^{\mathsf{A}} := [\![d \oplus b_3]\!]^{\mathsf{A}}$. This conversion requires sending $2k$ bits over two rounds. The final result can then be computed as $[\![ab]\!]^{\mathsf{A}} := a[\![b]\!]^{\mathsf{A}}$ where each party locally multiplies a by its share of b. Compared to performing the generic bit decomposition from Section 5.3, this approach reduces the round complexity and communication by $O(\log k)$.

Computing $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{B}} = [\![ab]\!]$ Once again, the bit injection procedure can be repeated here to convert $[\![b]\!]^{\mathsf{B}}$ to $[\![b]\!]^{\mathsf{A}}$ followed by computing $[\![a]\!]^{\mathsf{A}}[\![b]\!]^{\mathsf{A}}$ using the multiplication protocol.

5.5 Polynomial Piecewise Functions

This brings us to our final building block, the efficient computation of polynomial piecewise functions. These functions are constructed as a series of polynomials. Let $f_1, ..., f_m$ denote the polynomials with public coefficients and $-\infty = c_0 < c_1 < ... < c_{m-1} < c_m = \infty$ such that,

$$f(x) = \begin{cases} f_1(x), & x < c_1 \\ f_2(x), & c_1 \le x < c_2 \\ ... & \\ f_m(x), & c_{m-1} \le x \end{cases}$$

Our technique for computing f is to first compute a vector of secret shared values $b_1, ..., b_m \in \{0, 1\}$ such that $b_i = 1 \Leftrightarrow c_{i-1} < x \leq c_i$. f can then be computed as $f(x) = \sum_i b_i f_i(x)$.

Let us begin with the simple case of computing $[\![x]\!] < c$. This expression can then be rewritten as $[\![x]\!]^{\mathsf{A}} - c < 0$. Recall that x is represented as a two's complement value and therefore the most significant bit (MSB) of $[\![x - c]\!]$ denotes its sign, i.e. 1 iff $x - c < 0$. This implies that the inequality can be computed simply by extracting the MSB. This in turn can be computed by taking the Section 5.3 bit extraction of $[\![x - c]\!]$ to obtain binary shares of $[\![b]\!]^{\mathsf{B}} := [\![\mathsf{msb}(x - c)]\!]^{\mathsf{B}}$. When the bit-extraction is performed with binary secret sharing, the round complexity will be $O(\log k)$ while the communication is $O(k)$ bits. On the other hand, when the conversion is performed using a garbled circuit, the round complexity decreases to 1 with an increase in communication totaling $O(\kappa k)$ bits. Each b_i is the logical AND of two such shared bits which can be computed within the garbled circuit or by an additional round of interaction when binary secret sharing is used.

Each of the f_i functions are expressed as a polynomial $f_i([\![x]\!]) = a_{i,j}[\![x]\!]^j + ... + a_{i,1}[\![x]\!] + a_{i,0}$ where all $a_{i,l}$ are publicly known constants. When f_i is a degree 0 polynomial the computation $b_i f_i([\![x]\!])$ can be optimized as $a_{i,0}[\![b_i]\!]^{\mathsf{B}}$ using the techniques from Section 5.4. In addition, when the coefficients of f_i are integer, the computation of $a_{i,l}[\![x]\!]^l$ can be performed locally, given $[\![x]\!]^l$. However, when $a_{i,j}$ has a non-zero decimal, an interactive truncation will be performed as discussed in Section 5.1. The one exception to requiring a truncation is the case that f_i is degree 0 which can directly be performed using the techniques described above.

What remains is the computation of $[\![x]\!]^j, ..., [\![x]\!]^2$ given $[\![x]\!]$. The computation of these terms can be performed once and used across all $f_1, ..., f_m$ and requires $\log j$ round of communication. Importantly, the computation of these terms can be performed in parallel with the computation of the outer coefficients $b_1, ..., b_m$. As such, when computing these terms using binary secret sharing, the overall round complexity is unaffected and remains bounded by $O(\log k)$. However, in the event that garbled circuits are employed to compute the b_i terms, the round complexity decreases to $\log j \leq \log k$.

6 Machine Learning

Given the building blocks in Section 5, we show how to design efficient protocols for training linear regression, logistic regression and neural network models, on private data using the gradient descent method. We will discuss each model in detail next.

6.1 Linear Regression

Our starting point is the linear regression model using the stochastic gradient decent method. Given n training examples $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^D$ and the corresponding output variable $y_1, ..., y_n$, our goal is to find a vector $\mathbf{w} \in \mathbb{R}^D$ which minimizes the distance between $f(\mathbf{x}_i) := \mathbf{x}_i \cdot \mathbf{w} = \sum_j x_{ij} w_j$ and the true output y_i. There are many ways to define the distance between $f(\mathbf{x}_i)$ and y_i. In general, a cost function $C_{\{(\mathbf{x}_i, y_i)\}}(\mathbf{w})$ is defined and subsequently minimized in an optimization problem. We will focus on the commonly used L2 cost function, $C_{\{(\mathbf{x}_i, y_i)\}}(\mathbf{w}) := \frac{1}{n} \sum_i \frac{1}{2}(\mathbf{x}_i \cdot \mathbf{w} - y_i)^2$. That is, the squared difference between the predicted output $f(\mathbf{x}_i)$ and the true output y_i.

One reason this cost function is often used is due to it resulting in a straightforward minimization

problem. If we assume that there is a linear relation between $\mathbf{x}_i$ and y_i, the cost function C is convex which implies that the gradient decent algorithm will converge at the global minimum. The algorithm begins by initializing $\mathbf{w} \in \mathbb{R}^D$ with arbitrary values and at each step the gradient at $C(\mathbf{w})$ is computed. $\mathbf{w}$ is then updated to move down the gradient which decreases the value of $C(\mathbf{w})$. That is, at each iteration of the algorithm $\mathbf{w}$ is updated as $w_j := w_j - \alpha \frac{\partial C(\mathbf{w})}{\partial w_j} = w_j - \alpha \frac{1}{n} \sum_i^n (\mathbf{x}_i \cdot \mathbf{w} - y_i) x_{ij}$ The extra term α is known as the learning rate which controls how large of a step toward the minimum the algorithm should take at each iteration.

Batching One common optimization to improve performance is known as batching. The overall dataset of n examples are randomly divided into batches of size B denoted by $\mathbf{X}_1, ..., \mathbf{X}_{n/B}$ and $\mathbf{Y}_1, ..., \mathbf{Y}_{n/B}$. The update procedure for the jth batch is then defined as, $\mathbf{w} := \mathbf{w} - \alpha \frac{1}{B} \mathbf{X}_j^T \times (\mathbf{X}_j \times \mathbf{w} - \mathbf{Y}_j)$

Typically, once all the batches have been used once, the examples are randomly placed into new batches. Each set of batches is referred to as an epoch.

Learning Rate Choosing the correct learning rate α can be challenging. When α is too large the algorithm may repeatedly overstep the minimum and diverge. However, if α is too small the algorithm will take unnecessary many iterations to converge to the minimum. One solution is that α can be dynamically set using a variety of methods such as the one presented by Barzilai and Borwein [11]. A second option is to compute the cost function periodically on a subset of the training data. The value of α can then be dynamically tuned based on the rate at which the cost function is decreasing. For example, additively increase α by a small amount until the cost function increases at which point decrease it by some multiplicative factor. We leave the investigation of trade-offs between these methods to future work.

Termination Another problem is to determine when the algorithm should terminate. Sometimes an upper bound on the number of iterations is known. Alternatively, the cost function can also be used to determine the termination condition. When the cost function fails to decrease by a significant amount for several iterations, the algorithm can be concluded that the minimum has been reached.

Secure Linear Regression Implementing this algorithm in the secure framework of Section 5 is a relatively easy task. First, the parties jointly input the training examples $\mathbf{X} \in \mathbb{R}^{n \times D}, \mathbf{Y} \in \mathbb{R}^n$. We place no restrictions on how the data is distributed among the parties. For simplicity, the initial weight vector $\mathbf{w}$ is initialized as the zero vector. The learning rate α can be set as above.

In the secret shared setting the correct batch size, B, has several considerations. First, it should be large enough to ensure good quality gradients at each iteration. On the other hand, when B increases beyond a certain point, the quality of the gradient stops improving which results in wasted work and decreased performance. This trade-off has a direct consequence in the secret shared setting. The communication required for each iteration is proportional to B. Therefore, decreasing B results in a smaller bandwidth requirement. However, two rounds of interaction are required for each iteration of the algorithm, regardless of B. Therefore, we propose to set B proportional to the bandwidth available in the time required for one round trip (two rounds) or the minimum value of B determined by the training data, whichever is larger.

The batched update function can then be applied to each batch. The termination condition could then be computed periodically, e.g. every 100 batches. We note that this check need not add to the overall round

complexity. Instead, this check can be performed asynchronously with the update function. Moreover, due to it being performed infrequently, it will have little impact on the overall running time.

One important observation is that the two matrix multiplications performed in update function should be optimized using the delayed reshare technique of Section 5.2. This reduces the communication per multiplication to $B + D$ elements instead of $2DB$. In many cases the training data is very high dimensional, making this optimization of critical importance. The dominant cost of this protocol is 2 rounds of communication per iteration. In the semi-honest setting, each iteration sends $B + D$ shares per party and consumes $B + D$ truncation triples described in Section 5.1.

6.2 Logistic Regression

Logistic regression is a widely used classification algorithm and is conceptually similar to linear regression. The main difference is that the dependent variable y is binary as opposed to a real value in the case of linear regression. For example, given someone's credit card history $\mathbf{x}$, we wish to decide whether a pending transaction should be approved $y = 1$ or denied $y = 0$. In this case, the rate of convergence can be improved by bounding the output variable to be in the range between zero and one. This is achieved by applying an activation function f, which is bounded by zero and one, to the inner product, i.e. $y' = g(\mathbf{x}) = f(\mathbf{x} \cdot \mathbf{w})$. While there are many suitable activation functions, in the problem of logistic regression, f is defined to be the logistic function $f(u) = \frac{1}{1+e^u}$. One consequence of using this activation function is that the L2 cost function from the previous section is no longer convex. This is addressed by changing the cost function to be the cross-entropy equation, $C_{(\mathbf{x},y)}(\mathbf{w}) := -y \log f(\mathbf{x} \cdot \mathbf{w}) - (1 - y) \log(1 - f(\mathbf{x} \cdot \mathbf{w}))$. Given this, the update function for batch j can be defined as, $\mathbf{w} := \mathbf{w} - \alpha \frac{1}{B} \mathbf{X}_j^T \times (f(\mathbf{X}_j \times \mathbf{w}) - \mathbf{Y}_j)$. Observe that while the cost function has changed, the update function is quite similar to linear regression with the sole addition of the activation function f.

Unfortunately, computing the logistic function in the secret shared setting is an expensive operation. We instead follow the approach presented by Mohassel & Yupeng [48] where the logistic function is replaced with the piecewise function

$$f(x) = \begin{cases} 0, & x < -1/2 \\ x + 1/2, & -1/2 \leq x < 1/2 \\ 1, & 1/2 \leq x \end{cases}$$

As shown in [48, figure 7], the piecewise function roughly approximates the original. Moreover, [48] empirically showed that this change only decreases the accuracy of the MNIST model by 0.02 percent. However, we replaced the special purpose two party protocol that [48] presents with our general approach for computing any polynomial piecewise function that was detailed in Section 5.5. This allows us to easily handle other approximations of the logistic function too (e.g. non-linear piecewise polynomials, or [43] considers a piecewise linear function with 12 pieces).

6.3 Neural Nets

Neural network models have received a significant amount of interest over the last decade due to their extremely accurate predictions on a wide range of applications such as image and speech recognition. Conceptually, neural networks are a generalization of regression to support complex relationships between

high dimensional input and output data. A basic neural network can be divided up into m layers, each containing m_i nodes. Each node is a linear function composed with a non-linear "activation" function. To evaluate a neural network, the nodes at the first layer are evaluated on the input features. The outputs of these nodes are then forwarded as inputs to the next layer of the network until all layers have been evaluated in this manner. The training of neural networks is performed using back propagation in a similar manner to logistic regression except that each layer of the network should be updated in a recursive manner, starting at the output layer and working backward. Many different neural network activations functions have been considered in the literature. One of the most popular is the rectified linear unit (ReLU) function which can be expressed as $f(x) = \max(0, x)$. This function and nearly all other activations functions can easily be implemented using our piecewise polynomial technique from Section 5.5. Due to space constraints, we will only consider the evaluation of neural networks. However, we note that a single training iteration is approximately twice the cost of the evaluation. For a more detailed description of the exact operations, neural network evaluation entails, we refer readers to [51, 43].

7 Experiments

We demonstrate the practicality of our proposed framework with an implementation of training linear regression, logistic regression and neural network models and report on their efficiency. We defer a detailed explanation of the implemented machine learning algorithms to Appendix 6. The implementation was written in C++ and builds on the primitives provided by the libOTe library [52], and the linear algebra library Eigen [3]. All arithmetic shares are performed modulo 2^{64}. Binary shares employ high throughput vectorization techniques for matrix transpose and bit operations. That is, when computing a binary circuit on shared data it is almost always the case that the same circuit will be applied to several sets of inputs, e.g. the logistic function. In this case, 128 circuit evaluations can be compressed together by packing several shared bits into a single machine word and performing SIMD operations to that word. For more details on this technique, we refer to [10].

Due to the significant development time required to implement the maliciously secure protocols ([31] has no publicly available code), we have only implemented and report performance numbers for the semi-honest variant of our framework. This does not hinder comparison with prior work since they primarily focus on semi-honest protocols (in fact our work is the first maliciously secure protocol for machine learning).

Experimental Setup We perform all benchmarks on a single server equipped with 2 18-core Intel Xeon CPUs and 256GB of RAM. Despite having this many cores, each party performs the vast majority of their computation on a *single* thread. Using the Linux `tc` command we consider two network settings: a LAN setting with a shared 10Gbps connection and sub-millisecond RTT latency and a WAN setting with a 40Mbps maximum throughput and 40ms RTT latency. The server also employs hardware accelerated AES-NI to perform fast random number generation.

Datasets Our work is primarily focused on the performance of privacy-preserving machine learning solutions. As a result, we choose to use synthetic datasets which easily allow for a variable number of training examples and features and better demonstrate the performance of our training. We emphasize that we do NOT use synthetic data to measure accuracy of the train models. In fact, our training algorithms

(i.e. if our protocols were run honestly) are functionality equivalent to those of [48], and we refer the reader to their paper for precise accuracy measurements on real datasets.

More specifically, all function altering optimizations employed in our framework which may impact the accuracy of the trained models have already been proposed and discussed in prior work (e.g. [48]) and have been shown to have a very small impact on model accuracy compared to the original functions. Next, we elaborate in what ways our machine learning algorithms differ from textbook implementations.

1. *Fixed-point as opposed to floating point arithmetic.* In most cases, it is easy to compute an upper bound on any intermediate value and ensure that the fixed-point numbers contain sufficiently many bits to contain them. Moreover, a standard technique to improve the convergence rate of machine learning algorithms is to first normalize all of the features to be centered at zero and have constant standard deviations, e.g. 1. As such, the task of computing the upper bound on intermediate values is simplified due to all features having similar magnitudes.

2. Secret shared fixed-point error. As described in Section 5.1, multiplication of two secret shared fixed-point values can introduce an error of magnitude 2^{-d} or with very small probability an error with a large magnitude. As shown in [48, Figure 5], given sufficiently many decimal bits, this also has little to no impact on the accuracy of the trained models.

3. *Approximating the logistic function with a linear piecewise function.* A similar approximation algorithm was also employed by [48, Table 1] and was shown to result in a very small drop in accuracy. For example, the MNIST [6] handwriting recognition task has an accuracy of 98.62 percent as opposed to 98.64 percent with the true logistic function. The accuracy of the Arcene [1] was completely unaffected with an accuracy of 86 percent regardless of which activation function was used.

7.1 Linear Regression

We begin with the gradient decent protocol for learning linear regression models as detailed in Section 6.1. The computation of this protocol is easy given our framework. At each iteration, a random subset $\mathbf{X}_j$ of the dataset is sampled and the model is updated as $\mathbf{w} := \mathbf{w} - \alpha \frac{1}{B} \mathbf{X}_j^T \times (\mathbf{X}_j \times \mathbf{w} - \mathbf{Y}_j)$. We report performance in terms of iterations per second as opposed to end-to-end running time. This is primarily done to present the results in a way that can be easily generalized to other tasks. Figure 4 presents the throughput of our protocol compared to [48] and is further parameterized by the number of features $D \in \{10, 100, 1000\}$ and the size of the mini-batch $B \in \{128, 256, 512, 1024\}$.

The columns labeled "Online" denote the throughput of the input dependent computation while the columns labeled "Online + Offline" denote the total throughput including the pre-processing phase that is input independent. Our throughput is strictly better than that of [48]. In the LAN setting our online throughput is between 1.5 to 4.5 times greater than [48] which is primarily due to a more efficient multiplication protocol. For example, [48] requires preprocessed matrix beaver triples along with a more complex opening procedure. While our protocol's online throughput is considerably higher than [48], our main contribution is an offline phase that is orders of magnitude more efficient. Overall, our protocol becomes 200 to 1000 times greater than [48] due to the elimination of expensive beaver triples. The only operation performed in our offline phase is the generation of truncated shares $[\![r]\!], [\![r/2^d]\!]$ which requires computing the addition circuit which can be made extremely efficient.

In the WAN setting, our protocol is also faster than [48] by roughly a factor of 2 in the online phase and 10 to 1000 times faster when the overall throughput is considered. As before, the overall throughput of our protocol is almost identical to just the online phase, with a reduction in throughput of roughly 10 percent. This is in drastic contrast with [48] where the majority of the computation is performed in the offline phase.

Our protocol also achieves a smaller communication overhead compared to [48]. The communication complexity for the online phase of both protocols is effectively identical. Each party performs two matrix multiplications where shares of size B and D are sent. However, in the offline phase, [48] presents two protocols where the first requires $O(BD)$ exponentiations and $D + B$ elements to be communicated per iterations. Our protocol requires no exponentiations and achieves the same asymptotic communication overhead but with better constants. Due to a large number of exponentiations required by their protocol, [48] also propose a second technique based on oblivious transfer which is more computationally efficient at the expense of an increased communication of $O(BD\kappa)$ elements per iterations. In the LAN setting, the computationally efficient oblivious transfer protocol achieves the higher throughput. However, in the WAN setting, the communication overhead is the bottleneck and the exponentiation-based protocol becomes faster. In Figure 4, we always report and compare against the variant with the best throughput. In our protocol, on the other hand, the preprocessing is computationally more efficient than either approach presented by [48] and requires less communication.

Due to the offline phase of [48] having such a low throughput, the authors proposed an alternative client-aided protocol where semi-honest clients generate triplet shares in preprocessing and share them among the two servers. If we relabel an assisting client as the third sever, this variant of their protocol has a similar security model as ours with the notable exception that there is no natural way to extend it to the malicious setting. The advantage of adding a third party is that the throughput of the offline phase can be significantly improved. However, it is still several orders of magnitude slower than our preprocessing for a few reasons. First, their protocol requires that random matrices of the form $R_1 \times R_2 = R_3$ be generated by the third party, where R_1 is a $D \times B$ dimension matrix. These have to be constructed and sent to the two other parties resulting in high communication. On the other hand, our preprocessing simply requires the sending of $O(B + D)$ elements. Considering that D can be in the order of 100s this results in a significant reduction in computation and communication. Moreover, our overall protocol is already faster than the online phase of [48] and therefore is faster regardless of which preprocessing technique is used.

7.2 Logistic Regression

Our next point of comparison is with regards to the training of logistic regression models. This protocol is more complex compared to linear regression due to the need to compute the logistic function at each iteration. Our protocol approximates this using a piecewise linear function which requires switching to and from a binary secret sharing scheme. While relatively efficient computationally, it does have the negative consequence of increasing the round complexity of the protocol by 7 per iterations. In the LAN setting where latency is small, this has little impact. For example, given a batch size of $B = 128$ and dimension $D = 10$, our protocol can perform 2251 iterations per second using a single thread. Moreover, increasing the dimension to $D = 100$ only decreases the throughput to 1867 iterations per second. When compared to [48], this represents an order of magnitude improvement in running time. This difference is primarily attributed to [48] using garbled circuits which requires fewer rounds at the cost of increased bandwidth and

Setting	Dimension	Protocol	Batch Size B							
			Online				Online + Offline			
			128	256	512	1024	128	256	512	1024
LAN	10	This	11764	10060	7153	5042	11574	9803	6896	4125
		[48]	7889	7206	4350	4263	47	25	11	5.4
	100	This	5171	2738	993	447	5089	2744	1091	470
		[48]	2612	755	325	281	3.7	2.0	1.1	0.6
	1000	This	406	208	104	46	377	200	100	46
		[48]	131	96	45	27	0.44	0.24	0.12	0.06
WAN	10	This	24.6	24.5	24.3	23.9	20.8	20.7	20.6	20.3
		[48]	12.4	12.4	12.4	12.4	2.4	1.6	0.88	0.50
	100	This	24.5	24.1	23.7	23.3	20.7	20.4	20.1	19.4
		[48]	12.3	12.2	11.8	11.8	0.63*	0.37*	0.19*	0.11*
	1000	This	22.2	20.2	17.5	12.6	19.3	17.9	16.5	11.6
		[48]	11.0	9.8	9.2	7.3	0.06*	0.03*	0.02*	0.01*

Figure 4: Linear Regression performance measured in iterations per second (larger = better). Dimension denotes the number of features while batch size denotes number of samples used in each iteration. WAN setting has 40ms RTT latency and 40 Mbps throughput. The preprocessing for [48] was performed either using OT or the DGK cryptosystem with the faster protocol being reported above. The * symbol denotes that the DGK protocol was performed.

more expensive operations. For both linear and logistic regression, the offline phase is identical. As such, our extremely efficient offline phase results in a 200 to 800 times speedup over [48] for overall throughput.

In the WAN setting, our increased round complexity begins to degrade our performance to the point that [48] is almost as fast as our protocol during the online phase. For $B = 128$ and $D = 100$ our protocol performs 4.1 iterations per seconds while [48] achieves 3.1 iterations per second. However, as the batch size increases (resulting in a better rate of convergence), our protocol scales significantly better then [48]. Consider a batch size of $B = 1024$ where our protocol achieves 3.99 iterations per second while [48] achieves 0.99 iterations per seconds. When including the offline phase, our protocol receives almost no slowdown (5%) while [48] is between 2 and 100 times slower, representing 3 to 300 times improvement when compared with our protocol.

Our protocol also achieves a smaller communication overhead when approximating the logistic function. Primarily this is due to our protocol using a binary secret sharing and our new binary-arithmetic multiplication protocol from Section 5.4. In total, our protocol requires each party to send roughly $8Bk$ bits while [48], which uses garbled circuits, requires $1028Bk$ bits. The main disadvantage of our approach is that it requires 7 rounds of interaction compared to 4 rounds by [48]. However, at the cost of less than double the rounds, our protocol achieves a 128 times reduction in communication which facilitates a much higher throughput in the LAN or WAN setting when there is a large amount of parallelism.

7.3 Neural Networks

Our protocol particularly stands out when working with neural networks. The first network we consider (NN) is for the MNIST dataset and contains three fully connected layers consisting of 128, 128, and 10 nodes respectively. Between each layer, the ReLU activation function is applied using our piecewise polynomial technique. When *training* the NN network, our implementation is capable of processing 10 training iterations per seconds, with each iteration using a batch size of 32 examples. Proportionally, when using a batch size of 128, our protocol performs 2.5 iterations per second. An accuracy of 94% can be

Setting	Dimension	Protocol	Batch Size B							
			Online				Online + Offline			
			128	256	512	1024	128	256	512	1024
LAN	10	This	2251	2053	1666	1245	2116	1892	1441	1031
		[48]	188	101	41	25	37	20	8.6	4.4
	100	This	1867	1375	798	375	1744	1276	727	345
		[48]	183	93	46	24	3.6	1.9	1.1	0.6
	1000	This	349	184	95	42	328	177	93	41
		[48]	105	51	24	13.5	0.43	0.24	0.12	0.06
WAN	10	This	4.12	4.10	4.06	3.99	3.91	3.90	3.86	3.79
		[48]	3.10	2.28	1.58	0.99	1.4	0.94	0.56	0.33
	100	This	4.11	4.09	4.03	3.94	3.91	3.89	3.84	3.74
		[48]	3.08	2.25	1.57	0.99	0.52*	0.32*	0.17 *	0.01*
	1000	This	4.04	3.95	3.78	3.47	3.84	3.75	3.59	3.32
		[48]	3.01	2.15	1.47	0.93	0.06*	0.03*	0.02*	0.01*

Figure 5: Logistic Regression performance measured in iterations per second (larger = better). See caption of Figure 4.

achieved in 45 minutes (15 epochs). Compared to [48], with the same accuracy, our online running time is 80× faster while the overall running time is 55,000× faster.

We also consider a convolutional neural net (CNN) with 2 hidden layers as discussed [51]. This network applies a convolutional layer which maps the 784 input pixels to a vector of 980 features. Two fully connected layers with 100 and 10 nodes are performed with the ReLU activation function. For a detailed depiction, see [51, Figure 3]. For ease of implementation, we *overestimate* the running time by replacing the convolutional kernel with a fully connected layer. Our protocol can process 6 training iterations per second with a batch size of 32, or 2 iterations per second with a batch size of 128. We estimate, if the convolutional layer was fully implemented, that our training algorithm would achieve an equivalent accuracy as a plaintext model [51] of 99% in less than one hour of training time.

7.4 Inference

We also benchmark our framework performing machine learning inference using linear regression, logistic regression, and neural network models, as shown in Figure 6. For this task, a model that has already been trained is secret shared between the parties along with an unlabeled feature vector for which a prediction is desired. Given this, the parties evaluate the model on the feature vector to produce a prediction label. We note that inference (evaluation) for all three types of models can be seen as a special case of training (e.g. one forward propagation in case of neural networks) and hence can be easily performed using our framework. Following the lead of several prior works [48, 51, 44], we report our protocol's performance on the MNIST task [6] which takes $784 = 28 \times 28$ pixel images of handwritten numbers as input features and attempts to output the correct number. The accuracy of these models ranges from 93% (linear) to 99% (CNN).

When evaluating a linear model, our protocol requires exactly one online round of interaction (excluding the sharing of the input and reconstructing the output). As such, the online computation is extremely efficient, performing one inner product and communicating $O(1)$ bytes. The offline preprocessing, however, requires slightly more time at 3.7 ms along with the majority of the communication. The large difference between online and offline is primarily due to the fact that our offline phase is optimized for high throughput as opposed to low latency. Indeed, to take advantage of SSE vectorization instructions our offline phase

performs 128 times more work than is required. When compared to SecureML we observe that their total time for performing a single prediction is slightly less than ours due to their offline phase requiring one round of interaction as compared to our 64 rounds. However, achieving this running time in the two-party setting requires a very large communication of 1.6 MB as opposed to our (throughput optimized) 0.002 MB, an 800× improvement. Our protocol also scales much better as it requires almost the same running time to evaluate 100 predictions as it does 1. SecureML, on the other hand, incurs a 20× slowdown which is primarily in the communication heavy OT-based offline phase.

We observe a similar trend when evaluating a logistic regression model. The online running time of our protocols when evaluating a single input is just 0.2 milliseconds compared to SecureML requiring 0.7, with the total time of both protocols being approximately 4 milliseconds. However, our protocol requires 0.005 MB of communication compared to 1.6 MB by SecureML, a 320× difference. When 100 inputs are all evaluated together our total running time is 9.1ms compared to 54.2 by SecureML, a 6× improvement.

Our protocol requires 3ms in the online phase to evaluate the model and 8ms overall. SecureML, on the other hand, requires 193ms in the online phase and 4823ms overall, a 600× difference. Our protocol also requires 0.5 MB of communication as compared to 120.5 MB by SecureML.

More recently MiniONN [44] and Chameleon [51] have both proposed similar mixed protocol frameworks for evaluating neural networks. Chameleon builds on the two-party ABY framework [27] which in this paper we extend to the three-party case. However, Chameleon modifies that framework so that a semi-honest third party helps perform the offline phase as suggested in the client-aided protocol of [48]. As such, Chameleon's implementation can also be seen in the semi-honest 3 party setting (with an honest majority). In addition, because Chameleon is based on 2 party protocols, many of their operations are less efficient compared to this work and cannot be naturally extended to the malicious setting. MiniONN is in the same two-party model as SecureML. It too is based on semi-honest two-party protocols and has no natural extension to the malicious setting.

As Figure 6 shows, our protocol significantly outperforms both Chameleon and MiniONN protocols when ran on similar hardware. Our online running time is just 6 milliseconds compared to 1360 by Chameleon and 3580 by MiniONN. The difference becomes even larger when the overall running time is considered with our protocol requiring 10 milliseconds, while Chameleon and MiniONN respectively require 270× and 933× more time. In addition, our protocol requires the least communication of 5.2 MB compared to 12.9 by Chameleon and 657.5 by MiniONN. We stress that Chameleon's implementation is in a similar security model to us while MiniONN is in the two-party setting.

References

[1] Arcene data set. https://archive.ics.uci.edu/ml/datasets/Arcene. Accessed: 2016-07-14.

[2] Azure machine learning studio. https://azure.microsoft.com/en-us/services/machine-learning-studio/.

[3] Eigen library. http://eigen.tuxfamily.org/.

[4] Google cloud ai. https://cloud.google.com/products/machine-learning/.

[5] Machine learning on aws. https://aws.amazon.com/machine-learning/.

Model	Protocol	Batch Size	Running Time (ms) Online	Total	Comm. (MB)
Linear	This	1	0.1	3.8	0.002
		100	0.3	4.1	0.008
	SecureML [48]	1	0.2	2.6	1.6
		100	0.3	54.2	160
Logistic	This	1	0.2	4.0	0.005
		100	6.0	9.1	0.26
	SecureML [48]	1	0.7	3.8	1.6
		100	4.0	56.2	161
NN	This	1	3	8	0.5
	SecureML [48]	1	193	4823	120.5
CNN	This*	1	6	10	5.2
	Chameleon [51]	1	1360	2700	12.9
	MiniONN [44]	1	3580	9329	657.5

Figure 6: Running time and communication of privacy preserving inference (model evaluation) for linear, logistic and neural network models in the LAN setting (smaller = better). [48] was evaluated on our benchmark machine and [51, 44] are cited from [51] using a similar machine. The models are for the MNIST dataset with $D = 784$ features. NN denotes neural net with 2 fully connected hidden layers each with 128 nodes along with a 10 node output layer. CNN denotes a convolutional neural net with 2 hidden layers, see [51] details. * This work (over) approximates the cost of the convolution layers with an additional fully connected layer with 980 nodes.

[6] MNIST database. http://yann.lecun.com/exdb/mnist/. Accessed: 2016-07-14.

[7] Watson machine learning. https://www.ibm.com/cloud/machine-learning.

[8] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[9] Y. Aono, T. Hayashi, L. Trieu Phong, and L. Wang. Scalable and secure logistic regression via homomorphic encryption. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 142–144. ACM, 2016.

[10] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817. ACM, 2016.

[11] J. BARZILAI and J. J. Borwein. Two-point step size gradient methods. 8:141–148, 01 1988.

[12] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. pages 257–266.

[13] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. pages 192–206.

[14] D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security*, pages 57–64. Springer, 2012.

[15] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. Cryptology ePrint Archive, Report 2017/1114, 2017. https://eprint.iacr.org/2017/1114.

[16] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 486–497. ACM, 2007.

[17] R. Canetti. Security and composition of multiparty cryptographic protocols. 13(1):143–202, 2000.

[18] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

[19] N. Chandran, J. A. Garay, P. Mohassel, and S. Vusirikala. Efficient, constant-round and actively secure MPC: beyond the three-party case. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 277–294. ACM, 2017.

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

[21] M. Chase, R. Gilad-Bachrach, K. Laine, K. Lauter, and P. Rindal. Private collaborative neural network learning.

[22] K. Chaudhuri and C. Monteleoni. Privacy-preserving logistic regression. In *Advances in Neural Information Processing Systems*, pages 289–296, 2009.

[23] K. Chida, G. Morohashi, H. Fuji, F. Magata, A. Fujimura, K. Hamada, D. Ikarashi, and R. Yamamoto. Implementation and evaluation of an efficient secure computation system using rfor healthcare statistics. *Journal of the American Medical Informatics Association*, 21(e2):e326–e331, 2014.

[24] M. Chiesa, D. Demmler, M. Canini, M. Schapira, and T. Schneider. Towards securing internet exchange points against curious onlookers. In L. Eggert and C. Perkins, editors, *Proceedings of the 2016 Applied Networking Research Workshop, ANRW 2016, Berlin, Germany, July 16, 2016*, pages 32–34. ACM, 2016.

[25] B. Crypto. Spdz-2: Multiparty computation with spdz online phase and mascot offline phase, 2016.

[26] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation.

[27] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[28] W. Du and M. J. Atallah. Privacy-preserving cooperative scientific computations. In *csfw*, volume 1, page 273. Citeseer, 2001.

[29] W. Du, Y. S. Han, and S. Chen. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *SDM*, volume 4, pages 222–233. SIAM, 2004.

[30] M. K. Franklin, M. Gondree, and P. Mohassel. Multi-party indirect indexing and applications. pages 283–297.

[31] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, 2017.

[32] A. Gascon, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Secure linear regression on vertically partitioned datasets.

[33] I. Giacomelli, S. Jha, M. Joye, C. D. Page, and K. Yoon. Privacy-preserving ridge regression over distributed data from lhe. Cryptology ePrint Archive, Report 2017/979, 2017. https://eprint.iacr.org/2017/979.

[34] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[35] R. Gilad-Bachrach, K. Laine, K. Lauter, P. Rindal, and M. Rosulek. Secure data exchange: A marketplace in the cloud. Cryptology ePrint Archive, Report 2016/620, 2016. http://eprint.iacr.org/2016/620.

[36] D. Harris. A taxonomy of parallel prefix networks, 12 2003.

[37] E. Hesamifard, H. Takabi, and M. Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[38] G. Jagannathan and R. N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 593–599. ACM, 2005.

[39] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. pages 486–498.

[40] R. Kumaresan, S. Raghuraman, and A. Sealfon. Network oblivious transfer. In M. Robshaw and J. Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 366–396. Springer, 2016.

[41] J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens. Application-scale secure multiparty computation. In *European Symposium on Programming Languages and Systems*, pages 8–26. Springer, 2014.

[42] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Annual International Cryptology Conference*, pages 36–54. Springer, 2000.

[43] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 619–631. ACM, 2017.

[44] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–631. ACM, 2017.

[45] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. Pics: Private image classification with svm. Cryptology ePrint Archive, Report 2017/1190, 2017. https://eprint.iacr.org/2017/1190.

[46] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. Learning differentially private language models without losing accuracy. *arXiv preprint arXiv:1710.06963*, 2017.

[47] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: The garbled circuit approach. pages 591–602.

[48] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[49] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, 1999.

[50] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348. IEEE, 2013.

[51] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications.

[52] P. Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[53] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: Scalable provably-secure deep learning. *arXiv preprint arXiv:1705.08963*, 2017.

[54] A. P. Sanil, A. F. Karr, X. Lin, and J. P. Reiter. Privacy preserving regression modelling via distributed computation. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 677–682. ACM, 2004.

[55] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1310–1321. ACM, 2015.

[56] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 3–18. IEEE, 2017.

[57] A. B. Slavkovic, Y. Nardi, and M. M. Tibbits. " secure" logistic regression of horizontally and vertically partitioned distributed databases. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, pages 723–728. IEEE, 2007.

[58] C. Song, T. Ristenpart, and V. Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–601. ACM, 2017.

[59] S. Song, K. Chaudhuri, and A. D. Sarwate. Stochastic gradient descent with differentially private updates. In *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, pages 245–248. IEEE, 2013.

[60] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[61] J. Vaidya, H. Yu, and X. Jiang. Privacy-preserving svm classification. *Knowledge and Information Systems*, 14(2):161–178, 2008.

[62] S. Wu, T. Teruya, J. Kawamoto, J. Sakuma, and H. Kikuchi. Privacy-preservation for stochastic gradient descent application to secure logistic regression. *The 27th Annual Conference of the Japanese Society for Artificial Intelligence*, 27:1–4, 2013.

[63] H. Yu, J. Vaidya, and X. Jiang. Privacy-preserving svm classification on vertically partitioned data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 647–656. Springer, 2006.

[64] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. pages 220–250.

8 Proofs

8.1 Malicious Secure Fixed-Point Multiplication

Remarks: Observe that step 3 of the online phase need not be performed before step 4. Instead it can be performed in the following round and before any values dependent of the output $[\![z]\!]^{\mathsf{A}}$ are revealed. This observation allows the effective round complexity of the fixed-point multiplication protocol to be 1, conditioned on the next operation not to be revealing a value resulting from z.

Theorem 1. *Protocol* $\Pi_{\mathsf{mal-mult}}$ *of Figure 7 securely computes* $\mathcal{F}_{mult}$ *with abort, in the presence of one malicious party, where* $\mathcal{F}_{mult}$ *is defined as the functionality induced by* $\Pi_{\mathsf{mal-mult}}$ *in the semi-honest setting (steps 1a and 3 of the online phase can be omitted).*

proof sketch: The security of this protocol can easily be reduced to the security of [10]. First observe that the preprocessing phase makes only black box use of [31] and outputs a valid sharing of a uniformly distributed value $r' \in \mathbb{Z}_{2^k}$.

Parameters: A single 2-out-of-3 (or 3-out-of-3) share $[\![x']\!]^{\mathsf{A}} = (x'_1, x'_2, x'_3)$ over the ring $\mathbb{Z}_{2^k}$ and a integer $d < k$.
Preprocess:

1. All parties locally compute $[\![r']\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^k)$.
2. Define the sharing $[\![r]\!]^{\mathsf{B}}$ to be the $k-d$ most significant shares of $[\![r']\!]^{\mathsf{B}}$, i.e. $r = r'/2^d$.
3. The parties compute $[\![r'_2]\!]^{\mathsf{B}}, [\![r'_3]\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^k)$ and $[\![r_2]\!]^{\mathsf{B}}, [\![r_3]\!]^{\mathsf{B}} \leftarrow \mathsf{Rand}((\mathbb{Z}_2)^{k-d})$. r'_2, r_2 is revealed to party 1,2 and r'_3, r_3 to parties 2,3 using the $\mathsf{RevealOne}$ routine.
4. Using a ripple carry subtraction circuit, the parties jointly compute $[\![r'_1]\!]^{\mathsf{B}} := [\![r']\!]^{\mathsf{B}} - [\![r'_2]\!]^{\mathsf{B}} - [\![r'_3]\!]^{\mathsf{B}}$, $[\![r_1]\!]^{\mathsf{B}} := [\![r]\!]^{\mathsf{B}} - [\![r_2]\!]^{\mathsf{B}} - [\![r_3]\!]^{\mathsf{B}}$ and reveal r'_1, r_1 to parties 1,3.
5. Define the preprocessed shares as $[\![r']\!]^{\mathsf{A}} := (r'_1, r'_2, r'_3), [\![r]\!]^{\mathsf{A}} := (r_1, r_2, r_3)$.

Online: On input $[\![x]\!]^{\mathsf{A}}, [\![y]\!]^{\mathsf{A}}$,

1. The parties run the malicious secure multiplication protocol of [31, Protocol 4.2] where operations are performed over $\mathbb{Z}_{2^k}$. This includes:
   (a) Run the semi-honest multiplication protocol [31, Section 2.2] on $[\![x]\!]^{\mathsf{A}}, [\![y]\!]^{\mathsf{A}}$ to obtain a sharing of $[\![z']\!]^{\mathsf{A}} := [\![x]\!]^{\mathsf{A}}[\![y]\!]^{\mathsf{A}}$. $\oplus$ and $\wedge$ operations are replaced with $+, *$ respectively.
   (b) Before any shares are revealed, run the triple verification protocol of [31, Protocol 2.24] using $([\![x]\!]^{\mathsf{A}}, [\![y]\!]^{\mathsf{A}}, [\![z']\!]^{\mathsf{A}})$.
2. In the same round that party i sends z'_i to party $i+1$ (performed in step 1a), party i sends $(z'_i - r'_i)$ to party $i+2$.
3. Before any shares are revealed, party $i+1$ locally computes $(z'_i - r'_i)$ and runs $\mathsf{compareview}(z'_i - r'_i)$ with party $i+2$. If they saw different values both parties send $\perp$ to all other parties and abort.
4. All parties compute $(z' - r') = \sum_{i=1}^{3}(z_i - r'_i)$.
5. Output $[\![z]\!]^{\mathsf{A}} := [\![r]\!]^{\mathsf{A}} + (z' - r')/2^d$.

Figure 7: Single round share malicious secure fixed-point multiplication protocol $\Pi_{\mathsf{mal-mult}}$.

Now observe that sending $z'_i - r'_i$ reveals no information about x, y, z' due to r' being uniformly distributed. What remains to be shown is that if a malicious party i sends an incorrect value for $z'_i - r'_i$, the honest parties will abort before any private information is revealed. By the correctness of step 1b, party $i+1$ will hold the correct value of z_i. As such, party $i+1$ can compute the correct value of $z'_i + r'_i$ and run $\mathsf{compareview}(z'_i + r'_i)$ with party $i+2$. Conditioned on not aborting, both agree on the correct value of $z' + r'$. Note that $\mathsf{compareview}$ is general technique for ensuring both parties agree on the specified value. We refer interested readers to [31] for more details. Should the parties abort, no information is revealed due to the prohibition on revealing any private values before step 3 of the online phases completes. □