# Client-Aided Privacy-Preserving Machine Learning

Peihan Miao[1], Xinyi Shi[1], Chao Wu[2], and Ruofan Xu[3]

[1]Brown University, Providence, USA
[2]University of California, Riverside, USA
[3]University of Illinois Urbana-Champaign, Urbana, USA

**Abstract**

Privacy-preserving machine learning (PPML) enables multiple distrusting parties to jointly train ML models on their private data without revealing any information beyond the final trained models. In this work, we study the client-aided two-server setting where two non-colluding servers jointly train an ML model on the data held by a large number of clients. By involving the clients in the training process, we develop efficient protocols for training algorithms including linear regression, logistic regression, and neural networks. In particular, we introduce novel approaches to securely computing inner product, sign check, activation functions (e.g., ReLU, logistic function), and division on secret shared values, leveraging lightweight computation on the client side. We present constructions that are secure against semi-honest clients and further enhance them to achieve security against malicious clients. We believe these new client-aided techniques may be of independent interest.

We implement our protocols and compare them with the two-server PPML protocols presented in SecureML (Mohassel and Zhang, S&P'17) across various settings and ABY2.0 (Patra et al., Usenix Security'21) theoretically. We demonstrate that with the assistance of untrusted clients in the training process, we can significantly improve both the communication and computational efficiency by orders of magnitude. Our protocols compare favorably in all the training algorithms on both LAN and WAN networks.

**Keywords:** Privacy-Preserving Machine Learning, Secure Multi-Party Computation, Client-Aided Protocols.

## 1 Introduction

In recent years, we have witnessed machine learning (ML) emerge as one of the most influential technologies and rapidly expanding research domains. Its applications span a diverse spectrum, ranging from recommendation systems to self-driving cars, large language models, and even medical prediction and diagnosis. This is in part due to increasing amount of data being collected and available in the Big Data era. Meanwhile, as these machine learning algorithms and applications are deployed in various real-world scenarios, data privacy is becoming increasingly critical, especially in domains dealing with sensitive or confidential data such as healthcare, finance, and government. In cases where entities are hesitant or restricted from sharing their data due to privacy regulations, the significance of protecting data privacy is further emphasized.

Addressing these concerns, privacy-preserving machine learning (PPML) has become a crucial approach to training ML models in a distributed manner, which enables multiple distrusting parties

to collaboratively train ML models on their private data while maintaining data privacy. The most commonly considered setting in PPML, as proposed by Mohassel and Zhang [29], involves data owners (e.g., clients) secret sharing their data among two non-colluding parties (e.g., servers), who then jointly perform training on the secret-shared data.

At a high level, this approach can be conceptualized as two servers engaging in secure two-party computation to train the ML model on secret-shared data. Importantly, the servers learn nothing beyond the final trained model, ensuring the privacy of individual data points. Nevertheless, prior work [16, 21, 27, 29, 30, 32, 33] has overlooked the fact that the data was initially owned by the clients in the clear. In this work, we show that actively involving clients in the training process can yield significant improvements in both communication and computational efficiency of the overall protocol.

## 1.1 Our Contributions

We study two-server PPML training where the data is held by a large number of clients. Since the clients initially hold the training data in the clear, they can assist in certain computations based on their clear data to achieve better efficiency than computing on shared data. Additionally, we can leverage techniques from secure two-party computation with the assistance of an untrusted third party by treating the clients as the untrusted third party. This approach introduces a novel way of computing activation functions as well as division in the training algorithms, which proves to be much more efficient than the garbled circuit-based approaches commonly used in PPML. We believe these client-aided techniques may be of independent interest.

**Our Contributions.** In this work, we

- develop a new *client-aided inner product* protocol that enables a client and two servers to jointly compute the inner product of two private vectors $\langle \mathbf{x}, \mathbf{y} \rangle$, where $\mathbf{x}$ is secret shared among the two servers and $\mathbf{y}$ is held by the client;
- develop a series of *client-aided* protocols that, with the assistance of an untrusted client, allow two servers
  - to determine if their secret shared value is positive or not,
  - to compute activation functions (e.g., ReLU, logistic function) on their secret shared value, and
  - to compute divisions on their secret shared values (for softmax);
- put these techniques all together into PPML training protocols for linear regression, logistic regression, and neural networks, which are secure against *semi-honest* servers and clients;
- present techniques to enhance our security guarantees to protect against *malicious clients*;
- implement our protocols and demonstrate performance improvement compared with prior work.

**Experimental Results.** We implement our two-server PPML protocols for both semi-honest and malicious clients. We compare our performance with SecureML [29] in various settings and compare with the state-of-the-art ABY2.0 [30] theoretically (their code is not available). For linear regression, we achieve an improvement of $6.12-1047\times$ over [29] in the LAN setting and $3.63-73.5\times$ in the WAN setting. For logistic regression, we achieve an improvement of $4.85-723\times$ on LAN and $2.71-44.3\times$ on WAN. For neural networks, we achieve an improvement of $3.19\times$ on LAN and $3.92\times$

on WAN. When enhancing our security guarantees to malicious clients, we incur a small constant ($2.55 - 4.95\times$) overhead compared to our semi-honest variant. This is orders of magnitude more efficient than the OT- and LHE-based variants of [29]. We also give comprehensive comparisons for the communication costs as well as the offline/online efficiency. See Section 6 for more details.

### 1.2 Related Work

**Privacy-Preserving Machine Learning.** In the PPML domain, secure multi-party computation has been used for various ML algorithms such as decision trees [24], $k$-means clustering [5,15], and SVM classification [35,39]. However, these solutions are far from practical due to the high overheads that they incur. Mohassel and Zhang [29] introduced a practically-efficient PPML framework in the two-server setting. Since then, there has been a rich body of research in PPML that follows the same framework: data owners first secret share their data among two or more non-colluding parties who then perform training on the secret-shared data. Prior work has studied this problem in various settings, including secure training and inference, semi-honest and malicious security, with a focus on a small number of servers (e.g., two-server [16,21,27,29,30,32,33], three-server [8,20,21,28,31,36], and four-server [6,9,20]) where the adversary can corrupt at most one of them. In this work, we focus on the two-server setting for ML training, and we anticipate that the client-aided techniques developed here can be applied to ML inference.

**Federated Learning.** As a similar setting of PPML, federated learning (FL) [3,4,10,17,19,25] enables multiple entities (e.g., mobile devices) to collaboratively train a model under the coordination of a central server (e.g., service provider) while keeping the training data decentralized, protecting the privacy of the individual users. The two-server setting has also been studied in FL [1,10]. Most of the existing FL frameworks rely on a key building block known as *secure aggregation* [1,3,4,10], which protects clients' raw data (in particular, individual model updates) through secure aggregation. However, they reveal the global model updates, particularly the mini-batch stochastic gradient descent, to the central server(s) as well as all the clients. Recent work has shown that this framework is vulnerable to various privacy attacks [13,26,34,37,40]. As a side product of this work, we can apply client-aided PPML to two-server FL to enhance the privacy guarantee of FL, revealing only the final model to the central servers.

### 1.3 Roadmap

We give a high-level overview of our new techniques in Section 2. We provide preliminaries including the definitions of required cryptographic building blocks and machine learning algorithms in Section 3. In Section 4, we present our new client-aided protocols for inner product, sign check, activation functions, and division over secret shared values. In Section 5, we assemble the building blocks to present our client-aided PPML protocols that are secure against semi-honest clients. Additionally, we enhance the security guarantees to protect against malicious clients in Appendix A. Finally, we discuss our performance and experimental results in Section 6.

## 2 Technical Overview

During the training process, we keep the invariant that all the intermediate values (e.g., model parameters, clients' data, etc.) are additively secret shared among the two servers. Their secret

shares are only revealed to each other when the training process is finished and they would like to learn the final model. We discuss how to maintain the invariant for each type of operation in the training algorithms. First, addition is almost free, which can be computed locally by the servers. We discuss below how to deal with the operations that require more work, and how we can improve the efficiency by involving an untrusted client in the computation. We refer the reader to Section 3.3 for the ML algorithms we consider in this work.

**Client-Aided Inner Product.** One of the key steps in linear regression is to compute the inner product of two vectors, one vector $\mathbf{w}$ denoting the current model, and the other vector $\mathbf{x}$ denoting the client's data. In the existing PPML framework, the servers hold secret shares of both $\mathbf{w}$ and $\mathbf{x}$ and they perform a secure two-party computation protocol to compute secret shares of the inner product $\langle \mathbf{w}, \mathbf{x} \rangle$, e.g., by using Beaver multiplication triples [2] generated in an offline setup phase [29].

We notice that since $\mathbf{x}$ is entirely known to the client, she can compute a masked inner product with $\mathbf{x}$ and share the masking information with the two servers. This improves both the computation and communication between the servers. Moreover, it does not require heavy computation on the client side, nor does it require extra round of communication between the servers and the client. In particular, the client still sends secret shares of $\mathbf{x}$ to the servers, along with which she will send some extra masking values. We present the detailed protocol in Section 4.1.

When the vectors have dimension 1 (as a special case), this technique can be used to compute multiplication of a value shared among the servers with another value held by the client. This will be a key building block below.

**Client-Aided Activation Functions.** For logistic regression and neural networks, besides vector inner product (and more generally matrix multiplication), we also need to perform activation functions (e.g., logistic function, ReLU) on secret shared values. To do this, we need a way for the two servers to jointly determine whether a secret shared value is positive or not (we view the value as a two's complement representation). This is not an arithmetic operation, and the existing PPML frameworks [28–30] mainly rely on garbled circuits that compute the sum of two secret shared values to determine its highest order bit.

In this work, we present a new approach that utilizes a client as an untrusted third party. For two secret shares $[\![x]\!]_0$ and $[\![x]\!]_1$, the problem of determining if $[\![x]\!]_0 + [\![x]\!]_1 > 0$ is essentially a secure comparison problem, namely determining whether $[\![x]\!]_0 > -[\![x]\!]_1$. Instead of relying on garbled circuits [38], we reduce this problem to a special secure two-party computation problem, private set intersection cardinality (PSI-CA), via a certain encoding of the input values. In particular, each party generates a set of elements based on their input and they jointly compute the cardinality of the intersection of the two sets. $[\![x]\!]_0 > -[\![x]\!]_1$ iff the set intersection cardinality is 1, and $[\![x]\!]_0 \leq -[\![x]\!]_1$ iff the set intersection cardinality is 0. With the assistance of an untrusted third party (i.e., untrusted client), PSI-CA can be securely computed in an extremely efficient way requiring only symmetric-key cryptographic operations.

There are two issues in this approach. First, the existing client-aided PSI-CA protocols reveal the cardinality of the set intersection to either the client or one of the two servers. However, it is crucial that the result is never revealed to any party in our PPML protocols. We develop a new way to secret share the cardinality result between the client and the two servers. Another issue is that the reduction above only works if values are both positive or both negative. We observe that $[\![x]\!]_0$ and $[\![x]\!]_1$ have different signs with high probability throughout the training process, hence we can ensure the comparison is only between values of the same sign in our protocol.

To compute ReLU, we need to multiply the secret shared PSI-CA result with the secret shared value $x$. We can utilize the aforementioned client-aided inner product (with dimension 1) to efficiently compute the multiplication. Putting it all together, we present the client-aided ReLU protocol in Section 4.2. We further extend these ideas to the logistic function in Section 5.2.

**Client-Aided Division.** In neural network training, we additionally need to compute a softmax function on secret shared values. We use the MPC-friendly variant of it (see Section 3.3) which requires division of two secret shared values. We compute the quotient bit-by-bit sequentially starting from the most significant bit. In every step, we need to compare the current dividend with the divisor, which can be done using the client-aided sign check protocol described above. The secret shared output needs to be multiplied with the secret shared divisor, which can be done using the client-aided inner product with dimension 1. The protocol is presented in Section 4.4.

**Security Against Malicious Clients.** In the aforementioned client-aided protocols, it is critical that the clients are semi-honest, namely they follow the protocol description honestly while trying to extract more information from the protocol execution. This might not be a realistic assumption in practice. Hence we further enhance the security guarantees of our protocol to protect against malicious clients. The two main building blocks we need is the client-aided inner product and client-aided sign check. To ensure security against malicious clients, we leverage the cut-and-choose technique to verify that the results are computed correctly. See Appendix A for details. As it turns out, our malicious variant only incurs a small constant overhead compared to our semi-honest variant and is orders of magnitude more efficient than prior work (see Section 6.5).

# 3 Preliminaries

**Notation.** We use $\lambda, \sigma$ to denote the computational and statistical security parameters, respectively. We use $[\![v]\!]$ to denote an additive secret sharing of a value $v \in \mathbb{Z}_{2^\ell}$ between two servers $\mathsf{S}_0, \mathsf{S}_1$. In particular, server $\mathsf{S}_i$ ($i \in \{0,1\}$) holds $[\![v]\!]_i$ such that $v = [\![v]\!]_0 + [\![v]\!]_1$. To sample a random additive secret sharing of $v$, we use the notation $([\![v]\!]_0, [\![v]\!]_1) \leftarrow \mathsf{Sharing}(v)$. We use $\xleftarrow{\$}$ to denote random sampling from a uniform distribution. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For a vector $\mathbf{v}$, we use $\mathbf{v}[i]$ to denote the $i$-th element of the vector. By $\mathsf{negl}(\lambda)$ we denote a negligible function, i.e., a function $f$ such that $f(\lambda) < 1/p(\lambda)$ holds for any polynomial $p(\lambda)$ and sufficiently large $\lambda$.

**Fixed-Point Arithmetic.** Throughout our protocols, we follow the prior work [29, 30] to use the two's complement fixed-point representation to denote real numbers and keep at most $\ell_f$ bits in the fractional part for all intermediate values during the training process. In particular, we transform a real number $x$ (with at most $\ell_f$ bits in its fractional part) into an integer in $\mathbb{Z}_{2^\ell}$ by computing $x' = 2^{\ell_f} \cdot x$. Furthermore, we assume that all intermediate values have at most $\ell_w$ bits in the whole number part and that $\ell_w + \ell_f \ll \ell$ (this follows from prior work [29, 30]). To multiply two real numbers $x$ and $y$, we multiply $x' = 2^{\ell_f} \cdot x$ with $y' = 2^{\ell_f} \cdot y$ to obtain $z' = x' \cdot y' \in \mathbb{Z}_{2^\ell}$. Note that $z'$ has $2 \cdot \ell_f$ bits representing the fractional part of the product, so we truncate the least significant $\ell_f$ bits of $z'$ such that it has $\ell_f$ bits in the fractional part. Since we keep the invariant that all intermediate values are additively secret shared between the two servers and that $\ell_w + \ell_f \ll \ell$, we can truncate $z'$ by truncating its shares $[\![z']\!]_0$ and $[\![z']\!]_1$ locally on the two servers [29].

We use the function $\mathsf{RtoI}(x)$ to denote the function of transforming a real number $x$ to an integer in $\mathbb{Z}_{2^\ell}$, namely $\mathsf{RtoI}(x) = 2^{\ell_f} \cdot x$. We use the function $\mathsf{Trunc}(x')$ to denote the function of truncating an integer in $\mathbb{Z}_{2^\ell}$ by the lowest order $\ell_f$ bits, namely $\mathsf{Trunc}(x') = \lfloor x/2^{\ell_f} \rfloor$. When we

compare $x \in \mathbb{Z}_{2^\ell}$ with 0, we view $x$ as a two's complement representation and compare it with 0. When we divide $x$ by $y$, which are both positive real numbers represented in $\mathbb{Z}_{2^\ell}$, we compute the quotient by $\mathsf{Quotient}(x, y) := \lfloor x \cdot 2^{\ell_f}/y \rfloor \in \mathbb{Z}_{2^\ell}$.

### 3.1 Secure Multi-Party Computation

Secure multi-party computation (MPC) [14, 38] allows multiple parties, each holding a private input, to jointly compute a function on their private inputs without revealing anything beyond the output of the function. In this work, we consider MPC protocols for three parties with honest majority. In particular, the three parties are two servers and a client, where the adversary corrupts either the client or one of the two servers. We say an adversary is *semi-honest* if it follows the protocol description honestly while trying to extract more information from it, while a *malicious* adversary may arbitrarily deviate from the protocol specification. In our work, we assume both servers are semi-honest, and we consider both semi-honest and malicious clients. We follow the Universal Composition (UC) security definition of MPC, and refer the reader to [7] for details.

**Private Set Intersection.** Private set intersection (PSI) is a special secure two-party computation (2PC) protocol which allows two parties, each holding a private set of elements, to jointly compute the intersection of their sets without revealing any other information. In this work, we will be leveraging techniques from client-aided PSI [18, 22] where the two parties compute PSI (more specifically, the cardinality of the set intersection, PSI-CA) with the assistance of an untrusted client.

**Yao's Millionaires' Problem.** Yao's millionaires' problem [38] is another special secure 2PC protocol that allows two parties, each holding a private input value, to jointly compare the two values. In this works, we will be reduce this problem to PSI-CA [36].

### 3.2 Pseudorandom Function

A pseudorandom function (PRF) is a keyed function that can be computed efficiently (in polynomial time) but looks like a random function without knowledge of the key. In particular, let $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$ where $\lambda$ is the security parameter, and let $\mathcal{F} = \{f : \{0,1\}^n \to \{0,1\}^m\}$. We say $F$ is a PRF if for any probabilistic polynomial time (PPT) $\mathcal{A}$,

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[ \mathcal{A}^{F_k(\cdot)} = 1 \right] - \Pr_{f \xleftarrow{\$} \mathcal{F}} \left[ \mathcal{A}^{f(\cdot)} = 1 \right] \right| \leq \mathsf{negl}(\lambda).$$

### 3.3 Machine Learning Algorithms

In this section, we briefly review the ML algorithms considered in this work, including linear regression, logistic regression, and neural networks. We refer the reader to prior work [29, 30] on more details about these ML models. We consider a set of training data $\{\mathbf{x}_i, y_i\}_{i=1,\ldots,n}$. All the algorithms take the stochastic gradient descent (SGD) approach, which involves iteratively updating a target coefficient vector/matrix by following the gradient of a particular loss function evaluated on a random batch of training data. In the SGD method, we use $B$ to denote the batch size, $\alpha$ to denote the learning rate, $E$ to denote the number of epochs, $n$ to denote the size of training data, and define $t = \frac{E \cdot n}{B}$ as the number of iterations.

**Linear Regression.** In linear regression, we try to learn a coefficient vector $\mathbf{w}$ such that the following loss function is minimized: $\sum_{i=1}^{n} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i)^2$. Applying SGD to the linear loss function gives that we update $\mathbf{w}$ in each iteration according to the following expression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i=1}^{B} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i) \cdot \mathbf{x}_i.$$

**Logistic Regression.** The only difference between logistic regression and linear regression is that the logistic (Sigmoid) function $f(z) = \frac{1}{1+e^{-z}}$ is applied to the inner product $z = \langle \mathbf{w}, \mathbf{x}_i \rangle$, and the loss function needs to be adjusted accordingly so that the loss function is convex and SGD still works. The SGD update step in this case is identical to linear regression except for applying the logistic function to the inner product. In particular,

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i=1}^{B} (f(\langle \mathbf{w}, \mathbf{x}_i \rangle) - y_i) \cdot \mathbf{x}_i.$$

The above logistic function is not MPC-friendly, and we follow the approach of [29] by considering a piecewise linear function instead, which they demonstrated yields comparable accuracy in training. We refer the reader there for more details. In particular, we approximate the logistic function by

$$f(z) = \begin{cases} 0 & \text{if } z < -1/2 \\ z + 1/2 & \text{if } z \in [-1/2, 1/2] \\ 1 & \text{if } z > 1/2 \end{cases} .$$

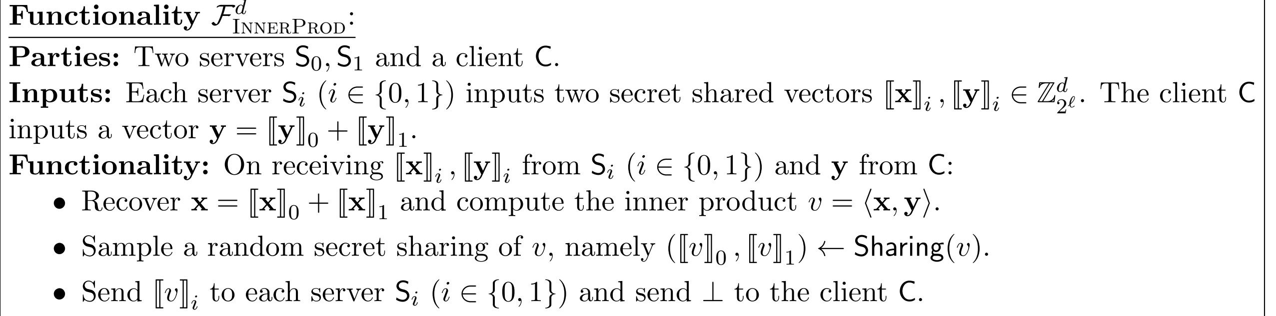
**Neural Networks.** Neural networks are a generalization of regression to learn more complex relationships between high dimensional input and output data. A basic neural network can be divided into $m$ layers, each containing $d_i$ nodes. Each node is a linear function composed with a non-linear *activation function*. One of the most popular activation functions considered in neural networks is the rectified linear unit (ReLU) function, which can be expressed as $f(x) = \max\{0, x\}$. 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. For classification problems with multiple classes, usually a *softmax* function is applied at the output layer, and we use the MPC-friendly variant [29] of the softmax function $f(u_i) = \frac{\text{ReLU}(u_i)}{\sum_{k=1}^{d_m} \text{ReLU}(u_k)}$. The training of neural networks is performed using SGD 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.

# 4 Client-Aided Protocols

## 4.1 Client-Aided Inner Product

In this section, we present a protocol for computing the inner product of a vector $\mathbf{x} \in \mathbb{Z}_{2^\ell}^d$ that is additively secret shared among two servers and another vector $\mathbf{y} \in \mathbb{Z}_{2^\ell}^d$ held by a client. As a result, the two servers learn an additive secret sharing of the inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ and the client learns nothing. The ideal functionality for our client-aided inner product is presented in Figure

1. Looking ahead, whenever we run this protocol, the vector $\mathbf{y}$ will also be shared among the servers (either directly shared by the client or learned from another protocol), hence in the ideal functionality we also let the servers input a secret share of $\mathbf{y}$. This will make this protocol better compile with our other protocols, especially in the case of malicious clients.



**Functionality $\mathcal{F}^d_{\textsc{InnerProd}}$:**

**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client $\mathsf{C}$.

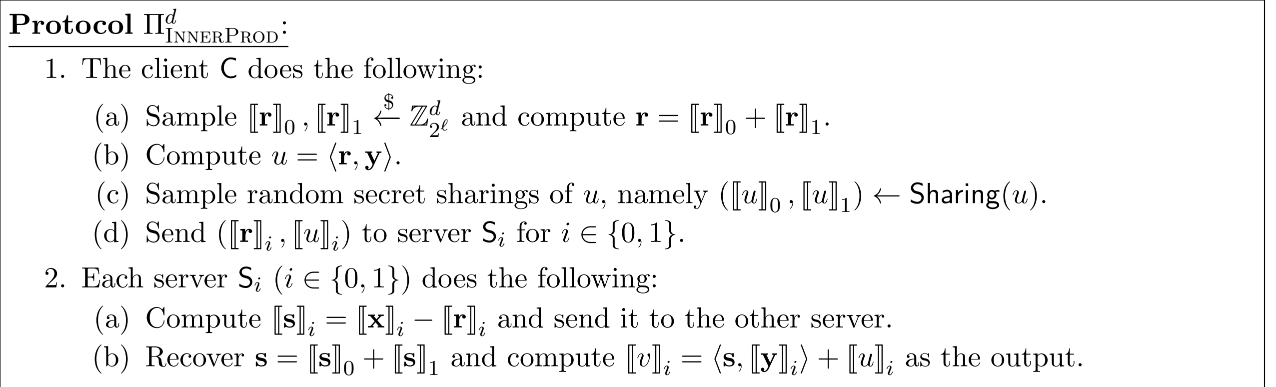
**Inputs:** Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) inputs two secret shared vectors $[\![\mathbf{x}]\!]_i, [\![\mathbf{y}]\!]_i \in \mathbb{Z}^d_{2^\ell}$. The client $\mathsf{C}$ inputs a vector $\mathbf{y} = [\![\mathbf{y}]\!]_0 + [\![\mathbf{y}]\!]_1$.

**Functionality:** On receiving $[\![\mathbf{x}]\!]_i, [\![\mathbf{y}]\!]_i$ from $\mathsf{S}_i$ ($i \in \{0,1\}$) and $\mathbf{y}$ from $\mathsf{C}$:

- Recover $\mathbf{x} = [\![\mathbf{x}]\!]_0 + [\![\mathbf{x}]\!]_1$ and compute the inner product $v = \langle \mathbf{x}, \mathbf{y} \rangle$.
- Sample a random secret sharing of $v$, namely $([\![v]\!]_0, [\![v]\!]_1) \leftarrow \mathsf{Sharing}(v)$.
- Send $[\![v]\!]_i$ to each server $\mathsf{S}_i$ ($i \in \{0,1\}$) and send $\perp$ to the client $\mathsf{C}$.

Figure 1: Ideal functionality $\mathcal{F}^d_{\textsc{InnerProd}}$ for computing the inner product.

**Construction Overview.** The client first samples a uniform random vector $\mathbf{r} \xleftarrow{\$} \mathbb{Z}^d_{2^\ell}$. Viewing $\mathbf{r}$ as a mask for the servers' input $\mathbf{x}$, the client generates a data-dependent multiplication triple by computing the inner product of $\mathbf{r}$ and its input vector $\mathbf{y}$, and sends the a secret share of the triple to the two servers. By using the data-dependent triple generated by the client, the two servers recover $\mathbf{x} - \mathbf{r}$ and compute a secret share of $\langle \mathbf{x}, \mathbf{y} \rangle$. Our protocol is described in Figure 2. We state the theorem below and give the security proof in Appendix B.1.



**Protocol $\Pi^d_{\textsc{InnerProd}}$:**

1. The client $\mathsf{C}$ does the following:
   (a) Sample $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1 \xleftarrow{\$} \mathbb{Z}^d_{2^\ell}$ and compute $\mathbf{r} = [\![\mathbf{r}]\!]_0 + [\![\mathbf{r}]\!]_1$.
   (b) Compute $u = \langle \mathbf{r}, \mathbf{y} \rangle$.
   (c) Sample random secret sharings of $u$, namely $([\![u]\!]_0, [\![u]\!]_1) \leftarrow \mathsf{Sharing}(u)$.
   (d) Send $([\![\mathbf{r}]\!]_i, [\![u]\!]_i)$ to server $\mathsf{S}_i$ for $i \in \{0,1\}$.
2. Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) does the following:
   (a) Compute $[\![\mathbf{s}]\!]_i = [\![\mathbf{x}]\!]_i - [\![\mathbf{r}]\!]_i$ and send it to the other server.
   (b) Recover $\mathbf{s} = [\![\mathbf{s}]\!]_0 + [\![\mathbf{s}]\!]_1$ and compute $[\![v]\!]_i = \langle \mathbf{s}, [\![\mathbf{y}]\!]_i \rangle + [\![u]\!]_i$ as the output.

Figure 2: Protocol $\Pi^d_{\textsc{InnerProd}}$ for computing the inner product.

**Theorem 4.1.** *The protocol $\Pi^d_{\textsc{InnerProd}}$ (Figure 2) securely computes the ideal functionality $\mathcal{F}^d_{\textsc{InnerProd}}$ (Figure 1) against a semi-honest adversary that corrupts either the client $\mathsf{C}$ or one of the two servers.*

**Communication and Optimizations.** In our protocol, each party computes only one inner product, so the servers and the client compute three inner products in total. The communication between the client and two servers is $(2d+2)$ ring elements in $\mathbb{Z}_{2^\ell}$ and the communication among the two servers is $2d$ ring elements. The total communication is $(4d+2)$ ring elements.

We discuss some optimizations in our implementation. In Steps 1a, 1c, 1d, the client needs to sample random vectors $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1$ as well as random secret shares of $u$, and send them to the servers, leading to a total communication cost of $(2d+2)$ ring elements in $\mathbb{Z}_{2^\ell}$. To reduce this communication, we let each server share a PRF key with the client. Then the client can use the

PRF keys to generate (pseudo)random vectors $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1$ for the two servers without communication. To generate a (pseudo)random secret sharing of $u$, the client can use the shared PRF key with one server $\mathsf{S}_0$ to generate $[\![u]\!]_0$ without communication, and send the other share $[\![u]\!]_1$ to $\mathsf{S}_1$. That is, apart from sharing the PRF keys, we can reduce the communication between the client and the servers from $(2d+2)$ to 1 ring element.

### 4.2 Client-Aided Sign Check

In this section, we present a protocol that allows two servers to jointly learn if a secret shared value $x \in \mathbb{Z}_{2^\ell}$ is positive or not (by viewing $x$ as a two's complement representation), with the assistance of a client. The three parties will learn a binary secret sharing of the sign check outcome $b$. In particular, the servers both learn one binary share $b^{\mathsf{S}}$ and the client learns the other share $b^{\mathsf{C}}$ such that $b^{\mathsf{S}} \oplus b^{\mathsf{C}} = b$. The ideal functionality is presented in Figure 3. Looking ahead, this protocol will be used in computing activation functions as well as divisions (for softmax). In our learning algorithms, we note that the absolute value of $x$ is significantly less than $2^\ell$ throughout the training process, hence $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs with overwhelming probability. In particular, we assume $x$ has at most $\ell_f$ bits in the fractional part and $\ell_w$ bits in the whole number part, and that $\ell_w + \ell_f \ll \ell$ (this follows from prior work [28–30]). Given that $([\![x]\!]_0, [\![x]\!]_1)$ is a uniformly random share of $x \in \mathbb{Z}_{2^\ell}$, the probability that $[\![x]\!]_0$ and $[\![x]\!]_1$ have the same sign is no greater than $2^{\ell_w+\ell_f-\ell}$. The proof follows from the analysis in [29]. Therefore, we assume $[\![x]\!]_0$ and $[\![x]\!]_1$ in the ideal functionality. In addition, we let the two servers learn a secret share of $b^{\mathsf{C}}$ so that this protocol can be incorporated more easily into other protocols.

**Functionality $\mathcal{F}_{\text{SignCheck}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client $\mathsf{C}$.
**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs a secret shared value $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$, where $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs. The client $\mathsf{C}$ has no input.
**Functionality:** On receiving $[\![x]\!]_i$ from $\mathsf{S}_i$ $(i \in \{0,1\})$:

- Recover $x = [\![x]\!]_0 + [\![x]\!]_1$ and let $b := (x > 0)$. That is, view $x$ as a two's complement representation and let $b$ be the indicator of weather $x$ is a positive number.
- Sample $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and let $b^{\mathsf{C}} := b \oplus b^{\mathsf{S}}$.
- Sample a random secret sharing of $b^{\mathsf{C}}$, namely $\left([\![b^{\mathsf{C}}]\!]_0, [\![b^{\mathsf{C}}]\!]_1\right) \leftarrow \mathsf{Sharing}(b^{\mathsf{C}})$.
- Send $b^{\mathsf{S}}$ to both servers and $[\![b^{\mathsf{C}}]\!]_i$ to each server $\mathsf{S}_i$ $(i \in \{0,1\})$. Send $b^{\mathsf{C}}$ to the client $\mathsf{C}$.

Figure 3: Ideal functionality $\mathcal{F}_{\text{SignCheck}}$ for determining if a secret shared value is positive or not.

**Construction Overview.** We first give an overview of our construction. The two servers hold an additive secret share of a value $x \in \mathbb{Z}_{2^\ell}$, namely each server $\mathsf{S}_i$ $(i \in \{0,1\})$ holds $[\![x]\!]_i$ such that $[\![x]\!]_0 + [\![x]\!]_1 = x$. We additionally assume that $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs. Suppose without loss of generality that $[\![x]\!]_0 \geq 0$ and $[\![x]\!]_1 < 0$, then checking whether $[\![x]\!]_0 + [\![x]\!]_1 > 0$ is equivalent to checking whether $[\![x]\!]_0 > -[\![x]\!]_1$, where both $[\![x]\!]_0$ and $-[\![x]\!]_1$ are non-negative values. We take inspiration from [23] to reduce our problem to PSI and then leverage techniques from client-aided PSI.

Let $a = \overline{a_\ell \cdots a_1}$ denote the binary representation of a non-negative value $a$. We denote its *0-encoding* by $\mathbf{P}_a^0 = \{\overline{a_\ell \cdots a_{i+1} 1 0 \cdots 0} | i \in [\ell], a_i = 0\}$ and its *1-encoding* by $\mathbf{P}_a^1 = \{\overline{a_\ell \cdots a_i 0 \cdots 0} | i \in$

$[\ell], a_i = 1\}$. Note that all binary strings in the sets have the same length $\ell$. We then pad the two sets with dummy elements to be of size $\ell$ each. Define two sets $\mathbf{G}_a^0$ and $\mathbf{G}_a^1$ as follows. $\mathbf{G}_a^0$ is a set of size $\ell - |\mathbf{P}_a^0|$ where all the elements are random $\ell$-bit strings starting with 10, and $\mathbf{G}_a^1$ is a set of size $\ell - |\mathbf{P}_a^1|$ where all the elements are random $\ell$-bit strings starting with 11. Let the corresponding *augmented 0-encoding* be defined as $\mathbf{A}_a^0 = \mathbf{P}_a^0 \cup \mathbf{G}_a^0$ and *augmented 1-encoding* be $\mathbf{A}_a^1 = \mathbf{P}_a^1 \cup \mathbf{G}_a^1$. Following the work [23], the set intersection $\mathbf{A}^1_{[\![x]\!]_0} \cap \mathbf{A}^0_{-[\![x]\!]_1}$ has size 1 if and only if $[\![x]\!]_0 > -[\![x]\!]_1$ and the intersection is empty otherwise. For the other case where $[\![x]\!]_0 < 0$ and $[\![x]\!]_1 \geq 0$, we simply swap the tasks of two parties and check whether $-[\![x]\!]_0 < [\![x]\!]_1$.

Now we reduce our problem to computing the size of the intersection of two private sets, namely PSI-CA. With the assistance of an untrusted client, we can utilize techniques from client-aided PSI-CA [18, 22]. Nevertheless, we need an additional security guarantee that the servers and the client only learn a binary secret share of the PSI-CA result.

We leverage the fact that the output of our PSI-CA can only be 0 or 1, and we randomly choose to compare either $[\![x]\!]_0 > -[\![x]\!]_1$ or $[\![x]\!]_0 < -[\![x]\!]_1$ . In particular, the servers randomly sample a bit $b^{\mathsf{S}}$ and flip the comparison if $b^{\mathsf{S}} = 1$. To be more specific, $\mathsf{S}_0$ generates an augmented $(1-b^{\mathsf{S}})$-encoding of $[\![x]\!]_0$ and $\mathsf{S}_1$ generates an augmented $b^{\mathsf{S}}$-encoding of $-[\![x]\!]_1$. Then they perform a client-aided PSI-CA protocol using a pseudorandom function (PRF). The client-aided sign check protocol is presented in Figure 4. We state the theorem below and give the security proof in Appendix B.2.

**Protocol $\Pi_{\textsc{SignCheck}}$:**

0. The two servers $\mathsf{S}_0$ and $\mathsf{S}_1$ agree on a computational security parameter $\lambda$ and a pseudorandom function $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^\lambda$.
1. $\mathsf{S}_0$ samples a random bit $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and random PRF key $k \xleftarrow{\$} \{0,1\}^\lambda$, and sends $(b^{\mathsf{S}}, k)$ to $\mathsf{S}_1$.
2. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ does the following:
   (a) View $[\![x]\!]_i$ as a two's complement representation. If $[\![x]\!]_i \geq 0$, then let $b_i := 1 \oplus b^{\mathsf{S}}$; otherwise let $[\![x]\!]_i := -[\![x]\!]_i$ and let $b_i := b^{\mathsf{S}}$.
   (b) Generate an augmented $b_i$-encoding of $[\![x]\!]_i$ as $\mathbf{A}_i$.
   (c) Apply the PRF $F_k$ to each element in $\mathbf{A}_i$ to obtain $\mathcal{T}_i = F_k(\mathbf{A}_i)$.
   (d) Randomly shuffle the elements in $\mathcal{T}_i$ and send the shuffled set $\widetilde{\mathcal{T}_i}$ to the client $\mathsf{C}$.
3. Upon receiving $\widetilde{\mathcal{T}_0}$ and $\widetilde{\mathcal{T}_1}$ from the two servers, the client $\mathsf{C}$ sets $b^{\mathsf{C}} = 0$ if $\widetilde{\mathcal{T}_0} \cap \widetilde{\mathcal{T}_1} = \emptyset$, and sets $b^{\mathsf{C}} = 1$ otherwise.
4. The client $\mathsf{C}$ samples a random secret sharing of $b^{\mathsf{C}}$, namely $\left([\![b^{\mathsf{C}}]\!]_0, [\![b^{\mathsf{C}}]\!]_1\right) \leftarrow \mathsf{Sharing}\left(b^{\mathsf{C}}\right)$, and sends $[\![b^{\mathsf{C}}]\!]_i$ to each server $\mathsf{S}_i$ $(i \in \{0,1\})$.
5. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ outputs $(b^{\mathsf{S}}, [\![b^{\mathsf{C}}]\!]_i)$. The client $\mathsf{C}$ outputs $b^{\mathsf{C}}$.

Figure 4: Protocol $\Pi_{\textsc{SignCheck}}$ for determining if a secret shared value is positive or not.

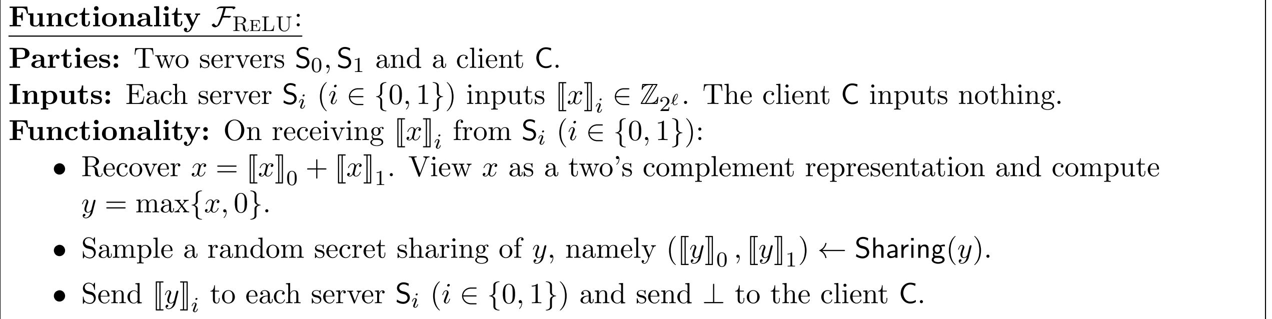
**Theorem 4.2.** *Assuming $F$ is a secure PRF, the protocol $\Pi_{\textsc{SignCheck}}$ (Figure 4) securely computes the ideal functionality $\mathcal{F}_{\textsc{SignCheck}}$ (Figure 3) against a semi-honest adversary that corrupts either the client $\mathsf{C}$ or one of the two servers.*

**Communication and Optimizations.** In our protocol, each server computes $\ell$ PRF opera-

tions. The total communication cost is $(2\lambda\ell + \lambda + 1)$ bits with 2 ring elements. We can apply the same optimization as in Section 4.1 to reduce communication by using shared PRF keys to generate random values. In particular, in Step 1 the servers can use a shared PRF key to generate (pseudo)random values $(b^{\mathsf{S}}, k)$ together without communication; in Step 4 the client $\mathsf{C}$ can use the shared PRF key to generate a (pseudo)random value with one server without communication. Then, the communication can be reduced to $2\lambda\ell$ bits with 1 ring elements.

### 4.3 Client-Aided ReLU

In this section, we present a protocol that allows two servers to jointly compute the ReLU function of an integer $x \in \mathbb{Z}_{2^\ell}$ that is additively secret shared among them, with the assistance of the client. Looking ahead, this protocol is a crucial component in computing activation functions. The ideal functionality for our client-aided ReLU is presented in Figure 5.


**Functionality $\mathcal{F}_{\text{ReLU}}$:**

**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client $\mathsf{C}$.

**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$. The client $\mathsf{C}$ inputs nothing.

**Functionality:** On receiving $[\![x]\!]_i$ from $\mathsf{S}_i$ $(i \in \{0,1\})$:

- Recover $x = [\![x]\!]_0 + [\![x]\!]_1$. View $x$ as a two's complement representation and compute $y = \max\{x, 0\}$.
- Sample a random secret sharing of $y$, namely $([\![y]\!]_0, [\![y]\!]_1) \leftarrow \mathsf{Sharing}(y)$.
- Send $[\![y]\!]_i$ to each server $\mathsf{S}_i$ $(i \in \{0,1\})$ and send $\perp$ to the client $\mathsf{C}$.

Figure 5: Ideal functionality $\mathcal{F}_{\text{ReLU}}$ for computing the ReLU function.

**Construction Overview.** The servers hold a secret share of an integer $x \in \mathbb{Z}_{2^\ell}$ and want to jointly learn a secret share of $\text{ReLU}(x) = \max\{0, x\} = (x > 0) \cdot x$. Observe that the ReLU function simply consists of a sign check operation and a multiplication operation, which can be computed by using the protocols in Sections 4.2 and 4.1, respectively. To combine these two protocols, the challenge is that the output of the sign check is a binary share among the servers and the client, while the input of the inner product should be additive secret shares. Observe that $x \cdot (b^{\mathsf{S}} \oplus b^{\mathsf{C}}) = x \cdot b^{\mathsf{S}} + (x \cdot b^{\mathsf{C}}) \cdot (1 - 2b^{\mathsf{S}})$ and the servers have $b^{\mathsf{S}}$ in clear, we only need to use the inner product protocol (with dimension 1) to compute a secret share of $x \cdot b^{\mathsf{C}}$, and then let each server $\mathsf{S}_i$ $(i \in \{0,1\})$ compute $[\![x \cdot (b^{\mathsf{S}} \oplus b^{\mathsf{C}})]\!]_i = [\![x]\!]_i \cdot b^{\mathsf{S}} + [\![x \cdot b^{\mathsf{C}}]\!]_i \cdot (1 - 2b^{\mathsf{S}})$. Our protocol is described in Figure 6. We state the theorem below and defer the proof to Appendix B.3.

**Theorem 4.3.** *The protocol $\Pi_{\text{ReLU}}$ (Figure 6) securely computes the ideal functionality $\mathcal{F}_{\text{ReLU}}$ (Figure 5) in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^{1}_{\text{InnerProd}})$-hybrid model against a semi-honest adversary that corrupts either the client $\mathsf{C}$ or one of the two servers.*

**Communication and Optimization.** The communication of a ReLU function consists of the communication of a sign check and an inner product of vectors with dimension 1. Applying the optimizations we mentioned, the communication between the client and the servers is $2\lambda\ell$ bits with 2 ring elements and the communication between the two servers is 2 ring elements. The computational cost on each server mainly contains $\ell$ PRF (AES) operations.

**Protocol $\Pi_{\text{ReLU}}$:**

1. $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}$ call $\mathcal{F}_{\text{SignCheck}}$ to compute

$$\left(\left(b^{\mathsf{S}}, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_0\right), \left(b^{\mathsf{S}}, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_1\right), b^{\mathsf{C}}\right) \leftarrow \mathcal{F}_{\text{SignCheck}}(\left[\!\left[x\right]\!\right]_0, \left[\!\left[x\right]\!\right]_1, \perp).$$

2. $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}$ call $\mathcal{F}^1_{\text{InnerProd}}$ to compute

$$(\left[\!\left[\alpha\right]\!\right]_0, \left[\!\left[\alpha\right]\!\right]_1, \perp) \leftarrow \mathcal{F}^1_{\text{InnerProd}}\left(\left(\left[\!\left[x\right]\!\right]_0, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_0\right), \left(\left[\!\left[x\right]\!\right]_1, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_1\right), b^{\mathsf{C}}\right).$$

3. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ outputs $\left[\!\left[y\right]\!\right]_i = \left[\!\left[x\right]\!\right]_i \cdot b^{\mathsf{S}} + \left[\!\left[\alpha\right]\!\right]_i \cdot (1 - 2b^{\mathsf{S}})$.

Figure 6: Protocol $\Pi_{\text{ReLU}}$ for computing ReLU in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model.

### 4.4 Client-Aided Division

In this section, we present a client-aided protocol that computes division of two shared values. Assume the servers hold secret shares of $x, y \in \mathbb{Z}_{2^\ell}$ such that $0 \le x \le y$ and $y \ne 0$, they jointly compute an additive secret share of the quotient $\mathsf{Quotient}(x, y) = \lfloor x \cdot 2^{\ell_f}/y \rfloor$ with the assistance of the client. See Figure 7 for the ideal functionality. Looking ahead, the division protocol is used to approximate the softmax function in the output layer of neural networks.

**Functionality $\mathcal{F}_{\text{Div}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client $\mathsf{C}$.
**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs two secret shared values $\left[\!\left[x\right]\!\right]_i \in \mathbb{Z}_{2^\ell}$ and $\left[\!\left[y\right]\!\right]_i \in \mathbb{Z}_{2^\ell}$ subject to the constraint that $0 \le x \le y$ and $y \ne 0$. The client $\mathsf{C}$ inputs nothing.
**Functionality:** On receiving $\left[\!\left[x\right]\!\right]_i$ and $\left[\!\left[y\right]\!\right]_i$ from $\mathsf{S}_i$ $(i \in \{0,1\})$:

- Recover $x = \left[\!\left[x\right]\!\right]_0 + \left[\!\left[x\right]\!\right]_1$, $y = \left[\!\left[y\right]\!\right]_0 + \left[\!\left[y\right]\!\right]_1$ and compute the quotient $q = \mathsf{Quotient}(x, y)$.
- Sample a random secret sharing of $q$, namely $(\left[\!\left[q\right]\!\right]_0, \left[\!\left[q\right]\!\right]_1) \leftarrow \mathsf{Sharing}(q)$.
- Send $\left[\!\left[q\right]\!\right]_i$ to each server $\mathsf{S}_i$ $(i \in \{0,1\})$ and send $\perp$ to the client $\mathsf{C}$.

Figure 7: Ideal functionality $\mathcal{F}_{\text{Div}}$ for computing division of two secret shared values.

**Construction Overview.** Inspired by the division protocol of SecureNN [36], we compute the quotient bit by bit. Let $k_i$ $(i \in \{\ell_f, \ldots, 0\})$ be every bit of the quotient. In our protocol, the servers compute a secret share of each bit step by step and then combine them together to get a secret share of the quotient. In particular, the servers store a secret share of an intermediate variable $u \in \mathbb{Z}_{2^\ell}$ (the dividend) initiated to be $x$. We start with the most significant bit $k_{\ell_f}$ by computing the sign check of $u - y$. Afterwards, we replace $u$ by $u = 2 \cdot (u - k_{\ell_f} \cdot y)$. Then we can compute the next bit in exactly the same way, i.e., $k_{\ell_f - 1}$ is equal to the sign of $u - y$. We can simply repeat the above two steps for the remaining bits. The main idea is that when we compute the $i$-th bit $k_i$ $(i \in \{\ell_f - 1, \ldots, 0\})$ after getting $k_{\ell_f}, \ldots, k_{i+1}$, we are actually computing the sign of $x \cdot 2^{\ell_f} - y \cdot \sum_{j=i+1}^{\ell_f} k_j \cdot 2^j - y \cdot 2^i$. Our protocol is described in Figure 8. We state the theorem below and give the security proof in Appendix B.4.

**Protocol $\Pi_{\text{Div}}$:**

1. Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) sets $[\![u_{\ell_f+1}]\!]_i = [\![x]\!]_i$.
2. For $j$ from $\ell_f$ downto 0:
   (a) Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) computes $[\![z_j]\!]_i = [\![u_{j+1}]\!]_i - [\![y]\!]_j + i$.
   (b) $\mathsf{S}_0$, $\mathsf{S}_1$ and $\mathsf{C}$ call $\mathcal{F}_{\text{SignCheck}}$ to compute
   $$\left(\left(b_j^{\mathsf{S}}, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_0\right), \left(b_j^{\mathsf{S}}, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_1\right), b_j^{\mathsf{C}}\right) \leftarrow \mathcal{F}_{\text{SignCheck}}([\![z_j]\!]_0, [\![z_j]\!]_1, \perp).$$
   (c) $\mathsf{S}_0$, $\mathsf{S}_1$ and $\mathsf{C}$ call $\mathcal{F}^1_{\text{InnerProd}}$ to compute
   $$([\![v_j]\!]_0, [\![v_j]\!]_1, \perp) \leftarrow \mathcal{F}^1_{\text{InnerProd}}\left(\left([\![y]\!]_0, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_0\right), \left([\![y]\!]_1, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_1\right), b_j^{\mathsf{C}}\right).$$
   (d) Each server $\mathsf{S}_i$ computes
   $$[\![k_j]\!]_i = i \cdot b_j^{\mathsf{S}} + \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_i \cdot (1 - 2b_j^{\mathsf{S}}), \qquad [\![v_j^*]\!]_i = [\![y]\!]_i \cdot b_j^{\mathsf{S}} + [\![v_j]\!]_i \cdot (1 - 2b_j^{\mathsf{S}}),$$
   $$[\![u_j]\!]_i = 2 \cdot \left([\![u_{j+1}]\!]_i - [\![v_j^*]\!]_i\right).$$
3. Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) outputs $[\![q]\!]_i = \sum_{j=0}^{\ell_f} 2^j \cdot [\![k_j]\!]_i$.

Figure 8: Protocol $\Pi_{\text{Div}}$ for computing division in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model.

**Theorem 4.4.** *The protocol $\Pi_{\text{Div}}$ (Figure 8) securely computes the ideal functionality $\mathcal{F}_{\text{Div}}$ (Figure 7) in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model against a semi-honest adversary that corrupts either the client $\mathsf{C}$ or one of the two servers.*

**Communication.** Considering all the computation among secret shared values, the servers and the client jointly compute $(\ell_f + 1)$ sign checks and multiplications in our division protocol. We can naturally use the protocols proposed in Sections 4.1 and 4.2. The total communication is $2(\ell_f + 1) \cdot \lambda \cdot \ell$ bits with $4 \cdot (\ell_f + 1)$ ring elements.

# 5 Client-Aided PPML

In this section, we present our client-aided two-server privacy-preserving machine learning protocols for linear regression, logistic regression, and neural networks. All the training algorithms follow the stochastic gradient descent (SGD) method, and we present a single ideal functionality $\mathcal{F}_{\text{ML_SGD}}$ in Figure 9.

## 5.1 Linear Regression

In this section, we present a protocol for an SGD iteration in linear regression training, where servers update the model $\mathbf{w}$ with the assistance of the clients in the mini-batch:

$$\mathbf{w}' \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i=1}^{B} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i) \cdot \mathbf{x}_i.$$

**Functionality $\mathcal{F}_{\text{ML_SGD}}$:**

**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a set of clients $\mathsf{C}_1, \ldots, \mathsf{C}_m$.

**Inputs:** Each client $\mathsf{C}_j$ ($j \in [m]$) inputs a data point $(\mathbf{x}_j, y_j)$ where $\mathbf{x}_j \in \mathbb{Z}_{2^\ell}^d$ and $y_j \in \mathbb{Z}_{2^\ell}$. The servers have no input.

**Functionality:** On receiving $(\mathbf{x}_j, y_j)$ from $\mathsf{C}_j$ ($j \in [m]$):

- Initialize the model $\mathbf{w}$ randomly.
- In each SGD iteration:
  - Pick a mini-batch of $B$ clients (public information).
  - Use these clients' data to update the model and obtain a new model $\mathbf{w}'$.
  - Sample a random secret sharing of $\mathbf{w}'$, namely $(\llbracket \mathbf{w}' \rrbracket_0, \llbracket \mathbf{w}' \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{w}')$.
  - Send $\llbracket \mathbf{w}' \rrbracket_i$ to each server $\mathsf{S}_i$ ($i \in \{0,1\}$) and send $\perp$ to all the clients.

Figure 9: Ideal functionality $\mathcal{F}_{\text{ML_SGD}}$ for two-server PPML.

**Construction Overview.** The most important steps in the process are two multiplications, one in forward propagation and one in backward propagation. Both can be computed by utilizing the client-aided inner product methodology we proposed. The servers first compute $\langle \mathbf{w}, \mathbf{x}_i \rangle$ by simply using the client-aided inner product for vectors of dimension $d$. The multiplication in the backward propagation can be viewed as $d$ inner products for vectors of dimension 1. We need to truncate the results of the inner products since all elements in the vectors represent real numbers. The protocol is described in Figure 10.

**Theorem 5.1.** *The protocol $\Pi_{\text{LinearSGD}}$ (Figure 10) securely computes the ideal functionality $\mathcal{F}_{\text{ML_SGD}}$ (Figure 9) for linear regression in the $\mathcal{F}_{\text{InnerProd}}$-hybrid model against a semi-honest adversary that corrupts either one of the two servers $\mathsf{S}_0, \mathsf{S}_1$, or an arbitrary subset of the clients.*

**Proof Sketch.** First we prove correctness of the protocol. Note that in each iteration of linear regression, the model should be updated as $\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{j=1}^{B} (\langle \mathbf{w}, \mathbf{x}_j \rangle - y_j) \cdot \mathbf{x}_j$. For each client $\mathsf{C}_j$ in a mini-batch: in Step 1b the servers learn a secret sharing of $\langle \mathbf{w}, \mathbf{x}_j \rangle$ (without truncation); in Step 1d they learn a secret sharing of $\langle \mathbf{w}, \mathbf{x}_j \rangle - y_j$ (with truncation); in Step 1e they learn secret shares of $(\langle \mathbf{w}, \mathbf{x}_j \rangle - y_j) \cdot \mathbf{x}_j$, which are truncated and combined into a vector $\llbracket \mathbf{u}_j \rrbracket$ in Step 1f. The correctness of truncation for fixed-point arithmetic is proved in [29]. Finally, in Step 2 the servers perform a linear combination of the current model $\llbracket \mathbf{w} \rrbracket$ and the gradient descent $\{\llbracket \mathbf{u}_j \rrbracket\}_{j \in [B]}$ to obtain an updated secret shared model $\llbracket \mathbf{w}' \rrbracket$.

In terms of privacy, any corrupted client does not learn any information from the protocol because it does not receive any message. This also holds for an arbitrary subset of corrupted and colluding clients. For a corrupted server, it only receives secret shared values in Steps 1a, 1b, 1c, 1e, which information theoretically hides the clients' data. The formal proof is similar to the proofs of Theorems 4.3 and 4.4.

**Communication and Optimizations.** In our linear regression protocol, the communication among the servers is $4Bd$ ring elements and that between the servers and the clients is $3Bd + 3B$ ring elements. Hence the total communication for an SGD iteration is $7Bd + 3B$ ring elements.

To further improve the communication, we can take advantage of the batched training. Consider the inner product in the forward propagation (Step 1b). The communication between the servers can be reduced by a factor of $B$ if different clients share the same vectors $(\llbracket \mathbf{r} \rrbracket_0, \llbracket \mathbf{r} \rrbracket_1)$ in protocol

**Protocol $\Pi_{\textsc{LinearSGD}}$:**

1. For each client $\mathsf{C}_j$ ($j \in [B]$) in a mini-batch:
   (a) The client $\mathsf{C}_j$ samples a random secret sharing of $\mathbf{x}_j$, namely $(\llbracket \mathbf{x}_j \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{x}_j)$, and sends $\llbracket \mathbf{x}_j \rrbracket_i$ to $\mathsf{S}_i$ ($i \in \{0,1\}$).
   (b) $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}^d_{\textsc{InnerProd}}$ to compute
   $$(\llbracket v_j \rrbracket_0, \llbracket v_j \rrbracket_1, \perp) \leftarrow \mathcal{F}^d_{\textsc{InnerProd}}\left((\llbracket \mathbf{w} \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_0), (\llbracket \mathbf{w} \rrbracket_1, \llbracket \mathbf{x}_j \rrbracket_1), \mathbf{x}_j\right).$$
   (c) The client $\mathsf{C}_j$ samples a random secret sharing of $y_j$, namely $(\llbracket y_j \rrbracket_0, \llbracket y_j \rrbracket_1) \leftarrow \mathsf{Sharing}(y_j)$, and sends $\llbracket y_j \rrbracket_i$ to $\mathsf{S}_i$ ($i \in \{0,1\}$).
   (d) Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) computes $\left\llbracket y_j^* \right\rrbracket_i = \mathsf{Trunc}\left(\llbracket v_j \rrbracket_i\right) - \llbracket y_j \rrbracket_i$.
   (e) For each $k \in [d]$, the two servers $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}^1_{\textsc{InnerProd}}$ to compute
   $$\left(\left\llbracket u_j^k \right\rrbracket_0, \left\llbracket u_j^k \right\rrbracket_1, \perp\right) \leftarrow \mathcal{F}^1_{\textsc{InnerProd}}\left((\llbracket y_j^* \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_0[k]), (\llbracket y_j^* \rrbracket_1, \llbracket \mathbf{x}_j \rrbracket_1[k]), \mathbf{x}_j[k]\right).$$
   (f) Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) combines $\left\{\left\llbracket u_j^k \right\rrbracket_i\right\}_{k \in [d]}$ into a vector, namely $\llbracket \mathbf{u}_j \rrbracket_i \in \mathbb{Z}^d_{2^\ell}$ where $\llbracket \mathbf{u}_j[k] \rrbracket_i = \mathsf{Trunc}\left(\left\llbracket u_j^k \right\rrbracket_i\right)$.
2. Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) outputs $\llbracket \mathbf{w}' \rrbracket_i = \llbracket \mathbf{w} \rrbracket_i - \frac{\alpha}{B} \sum_{j=1}^{B} \llbracket \mathbf{u}_j \rrbracket_i$.

Figure 10: Protocol $\Pi_{\textsc{LinearSGD}}$ (in the $\mathcal{F}_{\textsc{InnerProd}}$-hybrid model) for a single SGD iteration of linear regression.

$\Pi^d_{\textsc{InnerProd}}$ (Figure 2). Thus servers can reconstruct the same $\mathbf{w} - \mathbf{r}$ for all the clients in Step 2a of $\Pi^d_{\textsc{InnerProd}}$. Notice that it does not leak any information about the clients' data. Although the clients use the same $\mathbf{r}$ to compute $u_j := \langle \mathbf{r}, \mathbf{x}_j \rangle$ (for $j = 1, \ldots, B$), they will be sent to servers in the form of arithmetic secret sharing. As mentioned above, we use PRF to generate $\mathbf{r}_i$ without communication. Thus in Step 1b of $\Pi_{\textsc{LinearSGD}}$ (Figure 10), the communication between the two servers is $2d$ ring elements and the communication between the clients and the servers is $B$ ring elements for each iteration.

Similarly, the optimization can also be applied in the backward propagation. For each $k \in [d]$, the two servers $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}^1_{\textsc{InnerProd}}$ in Step 1e. Observe that during this step, $y_j^*$ is the same for all the dimensions of $\mathbf{x}_j$, so the client $\mathsf{C}_j$ can use the same $(\llbracket r \rrbracket_0, \llbracket r \rrbracket_1)$ in the protocol $\Pi^1_{\textsc{InnerProd}}$ to mask $y_j^*$ for each $k \in [d]$ in order to reduce the communication between the two servers. That is, each client $\mathsf{C}_j$ can actually compute a triple for the product of an integer and a vector, $y_j^* \cdot \mathbf{x}_j$, instead of $d$ irrelevant inner products with dimension 1. As a result, in Step 1e of $\Pi_{\textsc{LinearSGD}}$ (Figure 10), the communication between the two servers is reduced to $2B$ ring elements and the communication between the clients and the servers is $B \cdot d$ ring elements.

As mentioned earlier, we can use PRF to reduce communication. In Steps 1a and 1c, the client needs to sample random secret sharings of $\mathbf{x}_j$ and $y_j$. This communication can be reduced from $(2d + 2)$ to $(d + 1)$ ring elements by using PRF.

Furthermore, in Steps 1a and 1c, the client needs to send the secret sharings of its data to the servers, which leads to a total communication of $B \cdot (d+1) \cdot t$ ring elements. In our implementation,

we let all the clients share this with the servers at the beginning, so each data sample is shared only once and reused across different epochs. The communication becomes $n \cdot (d+1)$ ring elements.

In summary, the total communication between the two servers can be reduced to $2(B+d) \cdot t$ ring elements, and the total communication between all the clients and the servers can be reduced to $n \cdot (d+1) + B \cdot d \cdot t + B \cdot t$ ring elements.

### 5.2 Logistic Regression

Compared to linear regression, logistic regression merely adds a logistic function in each iteration. We use the MPC-friendly logistic function from [29]:

$$f(z) = \begin{cases} 0 & \text{if } z < -1/2 \\ z + 1/2 & \text{if } z \in [-1/2, 1/2] \\ 1 & \text{if } z > 1/2 \end{cases}$$

We observe that this function can be computed via two ReLU functions. In particular,

$$\textsc{ReLU}(z + 1/2) = \begin{cases} 0 & \text{if } z < -1/2 \\ z + 1/2 & \text{if } z \geq -1/2 \end{cases}$$

$$\textsc{ReLU}(1 - \textsc{ReLU}(z + 1/2)) = \begin{cases} 1 & \text{if } z < -1/2 \\ 1/2 - z & \text{if } z \in [-1/2, 1/2] \\ 0 & \text{if } z > 1/2 \end{cases}$$

$$1 - \textsc{ReLU}(1 - \textsc{ReLU}(z + 1/2)) = f(z)$$

Our protocol is represented in Figure 11.

**Theorem 5.2.** *The protocol $\Pi_{\text{LogisticSGD}}$ (Figure 11) securely computes the ideal functionality $\mathcal{F}_{\text{ML_SGD}}$ (Figure 9) for logistic regression in the $(\mathcal{F}_{\text{InnerProd}}, \mathcal{F}_{\text{SignCheck}})$-hybrid model against a semi-honest adversary that corrupts either one of the two servers $\mathsf{S}_0, \mathsf{S}_1$, or an arbitrary subset of the clients.*

**Proof Sketch.** First we prove correctness of the protocol. Note that in each iteration of logistic regression, the model should be updated as $\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{j=1}^{B} \left( f(\langle \mathbf{w}, \mathbf{x}_j \rangle) - y_j \right) \cdot \mathbf{x}_j$. For each client $\mathsf{C}_j$ in a mini-batch: in Step 1b the servers learn a secret sharing of $v_j = \langle \mathbf{w}, \mathbf{x}_j \rangle$ (without truncation), which is truncated in Step 1c. Next they jointly perform $f(\cdot)$ on the shared value $[\![v_j]\!]$. In Step 1d they learn a secret sharing of $\textsc{ReLU}(v_j + 1/2)$; in Step 1e they learn a secret sharing of $\textsc{ReLU}(1 - \textsc{ReLU}(v_j + 1/2))$; in Step 1g they obtain a secret sharing of $1 - \textsc{ReLU}(1 - \textsc{ReLU}(v_j + 1/2)) - y_j$, namely $f(\langle \mathbf{w}, \mathbf{x}_j \rangle) - y_j$. Next, in Step 1h they learn secret shares of $(f(\langle \mathbf{w}, \mathbf{x}_j \rangle) - y_j) \cdot \mathbf{x}_j$, which are truncated and combined into a vector $[\![\mathbf{u}_j]\!]$ in Step 1f. Finally, in Step 2 the servers perform a linear combination of the current model $[\![\mathbf{w}]\!]$ and the gradient descent $\{[\![\mathbf{u}_j]\!]\}_{j \in [B]}$ to obtain an updated secret shared model $[\![\mathbf{w}']\!]$.

In terms of privacy, any corrupted client does not learn any information from the protocol because it does not receive any message. This also holds for an arbitrary subset of corrupted and colluding clients. For a corrupted server, it only receives secret shared values in Steps 1a, 1b, 1d, 1e, 1f, 1h, which information theoretically hides the clients' data. The formal proof is similar to the proofs of Theorems 4.3 and 4.4.

**Communication and Optimizations.** Since the logistic function simply contains two ReLU operations, the communication overhead compared to linear regression is $4B \cdot \lambda \cdot \ell$ bits with $8B$ ring elements for each SGD iteration.

**Protocol $\Pi_{\text{LogisticSGD}}$:**

1. For each client $\mathsf{C}_j$ $(j \in [B])$ in a mini-batch:

   (a) The client $\mathsf{C}_j$ samples a random secret sharing of $\mathbf{x}_j$, namely $(\llbracket \mathbf{x}_j \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{x}_j)$, and sends $\llbracket \mathbf{x}_j \rrbracket_i$ to $\mathsf{S}_i$ $(i \in \{0,1\})$.

   (b) $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}^d_{\text{InnerProd}}$ to compute

   $$(\llbracket v_j \rrbracket_0, \llbracket v_j \rrbracket_1, \perp) \leftarrow \mathcal{F}^d_{\text{InnerProd}}\left((\llbracket \mathbf{w} \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_0), (\llbracket \mathbf{w} \rrbracket_1, \llbracket \mathbf{x}_j \rrbracket_1), \mathbf{x}_j\right).$$

   (c) Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ truncates the output $\llbracket v_j \rrbracket_i := \mathsf{Trunc}\left(\llbracket v_j \rrbracket_i\right)$.

   (d) $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}_{\text{ReLU}}$ to compute

   $$(\llbracket \alpha_j \rrbracket_0, \llbracket \alpha_j \rrbracket_1, \perp) \leftarrow \mathcal{F}_{\text{ReLU}}(\llbracket v_j \rrbracket_0, \llbracket v_j \rrbracket_1 + \mathsf{Rtol}(1/2), \perp).$$

   (e) $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}_j$ call $\mathcal{F}_{\text{ReLU}}$ to compute

   $$(\llbracket \beta_j \rrbracket_0, \llbracket \beta_j \rrbracket_1, \perp) \leftarrow \mathcal{F}_{\text{ReLU}}(-\llbracket \alpha_j \rrbracket_0, \mathsf{Rtol}(1) - \llbracket \alpha_j \rrbracket_1, \perp).$$

   (f) The client $\mathsf{C}_j$ samples additive secret sharings of $y_j$, namely $(\llbracket y_j \rrbracket_0, \llbracket y_j \rrbracket_1) \leftarrow \mathsf{Sharing}(y_j)$, and sends $\left(\llbracket y_j \rrbracket_i\right)$ to $\mathsf{S}_i$ $(i \in \{0,1\})$.

   (g) Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ computes $\left\llbracket y^*_j \right\rrbracket_i = \mathsf{Rtol}(i) - \llbracket \beta_j \rrbracket_i - \llbracket y_j \rrbracket_i$.

   (h) For each $k \in [d]$, the two servers $\mathsf{S}_0, \mathsf{S}_1$ and the client $\mathsf{C}_j$ call $\mathcal{F}^1_{\text{InnerProd}}$ to compute

   $$\left(\left\llbracket u^k_j \right\rrbracket_0, \left\llbracket u^k_j \right\rrbracket_1, \perp\right) \leftarrow \mathcal{F}^1_{\text{InnerProd}}\left((\llbracket y^*_j \rrbracket_0, \llbracket \mathbf{x}_j \rrbracket_0[k]), (\llbracket y^*_j \rrbracket_1, \llbracket \mathbf{x}_j \rrbracket_1[k]), \mathbf{x}_j[k]\right).$$

   (i) Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ combines $\left\{\left\llbracket u^k_j \right\rrbracket_i\right\}_{k \in [d]}$ into a vector, namely $\llbracket \mathbf{u}_j \rrbracket_i \in \mathbb{Z}^d_{2^\ell}$ where $\llbracket \mathbf{u}_j[k] \rrbracket_i = \mathsf{Trunc}\left(\left\llbracket u^k_j \right\rrbracket_i\right)$.

2. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ outputs $\llbracket \mathbf{w}' \rrbracket_i = \llbracket \mathbf{w} \rrbracket_i - \frac{\alpha}{B} \sum_{j=1}^{B} \llbracket \mathbf{u}_j \rrbracket_i$.

Figure 11: Protocol $\Pi_{\text{LogisticSGD}}$ for a single update iteration of logistic regression in the $(\mathcal{F}_{\text{InnerProd}}, \mathcal{F}_{\text{SignCheck}})$-hybrid model.

### 5.3 Neural Networks

To train an $m$-layer neural network with $d_i$ $(i \in \{0, \ldots, m\})$ nodes in the $i$-th layer, the servers update the coefficients of all nodes in each SGD iteration. All techniques we proposed for linear and logistic regression naturally extend to support neural network training.

All the functions in forward and backward propagation include additions, multiplications (inner product), and activation functions. For the multiplications that involve clients' data, we utilize our client-aided inner product protocol. For the multiplications that do not involve clients' data, we let the clients generate Beaver multiplication triples [2], similarly as in the client-aided variant of [29]. To evaluate the activation function ReLU and its derivative, we can simply run the client-aided ReLU twice in each iteration. Finally, for the MPC-friendly softmax function $f(u_i) =$

$\frac{\text{ReLU}(u_i)}{\sum_{k=1}^{d_m} \text{ReLU}(u_k)}$, we first run the client-aided ReLU to compute secret shared values of $\text{ReLU}(u_k)$, sum up all the shares, and then perform the client-aided division.

**Optimizations.** After the first layer of the neural network, the input to the remaining layers is shared among the two servers and not held by the clients, so we use the same approach as the client-aided variant of [29] to implement the remaining layers. In each layer, the three multiplications we need to calculate are $\mathbf{X} \cdot \mathbf{W}$, $\mathbf{X}^\top \cdot \mathbf{G}$, and $\mathbf{G} \cdot \mathbf{W}^\top$. Notice that each matrix $\mathbf{X}, \mathbf{W}, \mathbf{G}$ is used twice, and the clients can use the same random matrix to mask the same matrix twice, which can halve the communication among the servers and reduce half of the PRF operations for generating masks.

Note that there is no need to run separate sign checks for ReLU and its partial derivative, since they are exactly the same comparison. The (secret shared) comparison result will be multiplied by two different values, so the clients should generate two multiplication triples.

# 6 Performance Evaluation

We implement our two-server PPML protocols for training algorithms including linear regression, logistic regression, and neural networks, against both semi-honest and malicious clients. We report our performance in comparison with SecureML [29] in the semi-honest model in this section and defer the performance against malicious clients to Section 6.5. We did not compare the concrete performance with the state-of-the-art ABY2.0 [30] because their code for ML training is not available, but we did theoretical comparisons with their work.

We did not compare our protocols to prior works on PPML with three or more non-colluding servers because we believe our model differs from theirs in several key aspects. While the clients in our model could be considered as an additional server, the requirements on them are much weaker. Specifically, in prior works with three or more non-colluding servers, all the servers jointly hold secret shares of all the intermediate values. They participate in every step of the computation throughout the entire protocol. Nevertheless, our approach does not require the clients to stay online or hold any secret state. In client-aided sign check, activation functions, and division protocols, each time the servers may choose an arbitrary client for assistance while other clients are offline. After each iteration, the client may completely go offline without having to keep any secret state. Furthermore, the clients initially hold their data in the clear, which can be leveraged in client-aided inner product.

## 6.1 Implementation Details

We implement our protocols in C++. The only cryptographic primitive we need is PRF, which is instantiated with AES. We set the computational security parameter $\lambda = 128$ and statistical security parameter $\sigma = 40$.

**Experiment Settings.** Our experiments are performed on a single Amazon Web Services (AWS) Elastic Compute Cloud (EC2) c4.8xlarge virtual machine with 18-core 2.9GHz Intel Xeon CPU and 60 GB of RAM which is the same as [29]. We simulate the network connection using the Linux `tc` command. For the experiments on a LAN network, we set the round-trip time (RTT) latency to be 0.34 ms and network bandwidth to be 8192 Mbps, same as [29]. For the experiments on a WAN network, we set the RTT latency to be 60 ms and the network bandwidth to be 60 Mbps.

**Dataset and Parameters.** In our experiments, we train our algorithms on the MNIST dataset [11], which contains images of handwritten digits from 0 to 9. Each training sample has 784 features representing $28 \times 28$ pixels in the image.

In our training protocols, we have the number of features $d = 784$; we set the mini-batch size $B = 128$ and the number of epochs $E = 2$ (all the samples are used twice in training). The total number of training samples $n$ varies between 10,240 and 100,352. The total number of training iterations is $t = \frac{E \cdot n}{B}$.

For fixed-point arithmetic, we set $\ell = 64, \ell_f = 13, \ell_w = 6$. That is, all the values are represented in $\mathbb{Z}_{2^{64}}$, where the lowest order 13 bits are the fractional part, and we assume there are at most 32 bits in the whole number part. These parameters are taken from [29].

**Offline vs. Online.** In the protocols of [29], there is an offline and an online phase, where the offline phase includes all the computation and communication that can be done without presence of data while the online phase consists of all *data-dependent* steps of the protocols. In the offline phase, they proposed three different approaches, one based on oblivious transfer (OT), one based on linearly homomorphic encryption (LHE), and one client-aided. The OT-based and LHE-based offline protocols are performed among the two servers to generate multiplication triples, while the client-aided offline protocol relies on a client to generate the triples.

Our protocols, on the other hand, only have an online phase, where the clients are heavily involved in the protocol execution. In particular, they will generate *data-dependent* triples in the client-aided inner product protocol. Getting rid of the offline phase allows us to reduce the offline storage on the servers as well as the amount of communication between the servers. In our experiments below, we give comprehensive comparisons to [29] in both offline and online phases.

### 6.2 Linear Regression

In this section, we compare the performance of our semi-honest linear regression training to [29] instantiated with an OT-based, LHE-based, or client-aided offline phase. We report the running time in both LAN and WAN settings in Table 1 and the communication costs in Table 2.

| | $n$ | Offline Time (s) | | Online Time (s) | | Total Time (s) | |
|---|---|---|---|---|---|---|---|
| | | LAN | WAN | LAN | WAN | LAN | WAN |
| Our work | 10,240 | **0** | **0** | **1.32** | **52.1** | **1.32** | **52.1** |
| | 100,352 | **0** | **0** | **13.2** | **505** | **13.2** | **505** |
| OT-based [29] | 10,240 | 266 | 3,733 | 2.42 | 57.8 | 268 | 3,791 |
| | 100,352 | 2,667 | 36,600* | 25.8 | 557 | 2,692 | 37,157* |
| LHE-based [29] | 10,240 | 1,414 | 1,435 | 2.42 | 57.8 | 1,416 | 1,493 |
| | 100,352 | 13,800* | 14,000* | 25.8 | 557 | 13,826* | 14,557* |
| Client-aided [29] | 10,240 | 4.69 | 94.9 | 3.39 | 94.4 | 8.08 | 189 |
| | 100,352 | 52.0 | 749 | 35.3 | 1,126 | 87.3 | 1,875 |

Table 1: Running time of semi-honest linear regression on LAN and WAN networks comparing our protocol to [29] instantiated with different offline approaches. * indicates estimated running time.

In Table 1, we report the running time for both the offline and online phases in [29] as well as the total time. Our protocol does not incur any offline cost, and our online phase is also more efficient as our computation overhead is lower. In particular, in the online phase we achieve $1.83 - 2.67\times$ improvement over [29] in the LAN setting and $1.11 - 2.23\times$ improvement in the WAN setting. For

| | $n$ | Offline Comm (MB) | | Online Comm (MB) | | | Total |
|---|---|---|---|---|---|---|---|
| | | $\mathsf{S}-\mathsf{S}$ | $\mathsf{C}-\mathsf{S}$ | $\mathsf{S}-\mathsf{S}$ | $\mathsf{C}-\mathsf{S}$ | Total | Comm |
| Our work | 10,240 | **0** | **0** | **2.23** | 185 | **187** | **187** |
| | 100,352 | **0** | **0** | **21.8** | 1,814 | **1,836** | **1,836** |
| OT-based [29] | 10,240 | 24,151 | 0 | 125 | **123** | 248 | 24,399 |
| | 100,352 | 236,607 | 0 | 1,224 | **1,204** | 2,428 | 239,034 |
| LHE-based [29] | 10,240 | 115 | 0 | 125 | **123** | 248 | 362 |
| | 100,352 | 1,120* | 0 | 1,224 | **1,204** | 2,428 | 3,548* |
| Client-aided [29] | 10,240 | 0 | 614 | 368 | **123** | 491 | 1,105 |
| | 100,352 | 0 | 6,016 | 3,609 | **1,204** | 4,813 | 10,829 |

Table 2: Communication cost of semi-honest linear regression comparing our protocol to [29] instantiated with different offline approaches. "$\mathsf{S}-\mathsf{S}$" and "$\mathsf{C}-\mathsf{S}$" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

the total running time (offline + online), we achieve $6.12-1047\times$ improvement in the LAN setting and $3.63-73.5\times$ improvement in the WAN setting.

In Table 2, we report both the communication between the two servers and the communication between the clients and servers, which are denoted by "$\mathsf{S}-\mathsf{S}$" and "$\mathsf{C}-\mathsf{S}$" respectively in the table. Again, our protocol does not incur any offline cost. In the online phase, our communication cost between the two servers is significantly lower than [29]. In particular, our $\mathsf{S}-\mathsf{S}$ online communication is $2(B+d)\cdot t$ ring elements. The $\mathsf{S}-\mathsf{S}$ online communication of the OT-based and LHE-based protocols in [29] is $2n\cdot d+2(B+d)\cdot t$ ring elements, and that of the client-aided variant is $2n\cdot d+2(Bd+B)\cdot t$ ring elements. Although our online communication between the clients and servers is higher than [29], the total communication is still much lower than [29]. In particular, in the online phase our $\mathsf{S}-\mathsf{S}$ communication achieves $56.1-165\times$ improvement over [29], and our total online communication achieves $1.32-2.63\times$ improvement. For the total communication (offline + online), we achieve $1.93-130\times$ improvement.

**Increasing Mini-Batch Size.** If we increase the mini-batch size $B$, we can achieve lower communication, and hence the performance also improves especially in the WAN setting. This is because some part of the communication grows with the number of iterations. If the number of epochs and $n$ remain the same and the mini-batch size is increased, then the number of iterations decreases and the communication is lowered as well. See Figure 12 for the performance of the online phase in the WAN setting with different mini-batch sizes. We only compare with the client-aided variant of [29] because they achieve the most comparable overall running time.

**Comparison with ABY2.0 [30].** We compare our performance with [30] theoretically as their code is not available. Since [30] uses the same OT-based and LHE-based multiplication triples generation as [29] in the offline phase, as seen in Tables 1 and 2, their offline time and communication are already much higher than our total time and communication.

Although not mentioned in their paper, we observe that a similar client-aided approach can be applied to [30] to improve the efficiency of the offline phase while introducing some overhead in the online phase. Nevertheless, we expect our work to still outperform [30] in that case. With a client-aided offline phase, the total communication (offline + online) of [30] is $(12Bd+6B+4d)\cdot t$ ring elements. If using all the optimizations we mentioned, its communication can be reduced to $n\cdot(3d+1)+(Bd+3B+2d)\cdot t$ ring elements. In comparison, our total communication is $n\cdot(d+1)+(Bd+3B+2d)\cdot t$ ring elements and our computation cost is roughly half of [30]. This

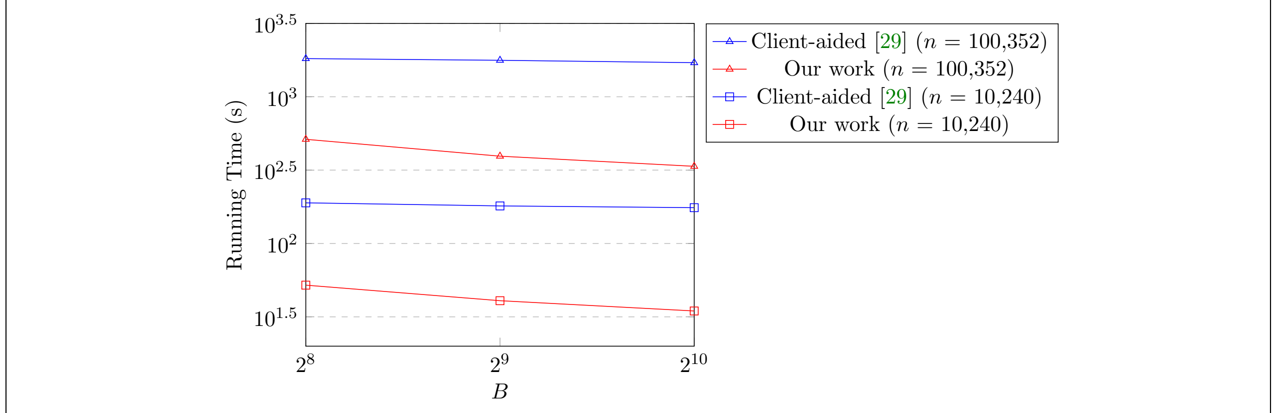



Figure 12: Total running time of semi-honest linear regression over WAN with different mini-batch sizes.

is because each server computes two matrix multiplications for a private inner product in [29, 30], while they each compute one matrix multiplication in our protocol.

### 6.3 Logistic Regression

In this section, we compare the performance of our semi-honest logistic regression training to [29] in Tables 3 and 4. Compared to linear regression, the only overhead of logistic regression is the cost of the activation function. In each iteration, each client and the two servers run $\Pi_{\mathrm{ReLU}}$ twice.

| | $n$ | Offline Time (s) | | Online Time (s) | | Total Time (s) | |
|---|---|---|---|---|---|---|---|
| | | LAN | WAN | LAN | WAN | LAN | WAN |
| Our work | 10,240 | **0** | **0** | **1.96** | **87.6** | **1.96** | **87.6** |
| | 100,352 | **0** | **0** | **19.8** | **850** | **19.8** | **850** |
| OT-based [29] | 10,240 | 266 | 3,733 | 3.86 | 108 | 270 | 3,841 |
| | 100,352 | 2,667 | 36,600* | 40.0 | 1,056 | 2,707 | 37,656* |
| LHE-based [29] | 10,240 | 1,414 | 1,435 | 3.86 | 108 | 1,418 | 1,543 |
| | 100,352 | 13,800* | 14,000* | 40.0 | 1,056 | 13,840* | 15,056* |
| Client-aided [29] | 10,240 | 4.71 | 95.4 | 4.81 | 142 | 9.52 | 237 |
| | 100,352 | 55.9 | 941 | 46.3 | 1,398 | 102 | 2,339 |

Table 3: Running time of semi-honest logistic regression on LAN and WAN networks comparing our protocol to [29] instantiated with different offline approaches. * indicates estimated running time.

Our computation cost of one ReLU mainly consists of $\ell$ PRF operations and sorting these $\ell$ PRF results for each server. As shown in Table 3 for the running time, in the online phase we achieve $1.97 - 2.45\times$ improvement over [29] in the LAN setting and $1.23 - 1.64\times$ improvement in the WAN setting. For the total running time (offline + online), we achieve $4.85 - 723\times$ improvement in the LAN setting and $2.71 - 44.3\times$ improvement in the WAN setting.

In terms of communication, our total communication overhead in $\Pi_{\mathrm{ReLU}}$ is $4B \cdot t \cdot \lambda \cdot \ell$ bits with $8B \cdot t$ ring elements, while the communication overhead in [29] is $2B \cdot t \cdot (2\lambda \cdot (2\ell - 1) + 3\ell)$ bits. In [30], the online communication for one ReLU is $3\ell + 230$ bits and the offline communication is $1337\lambda + 5\ell + 1332$ bits, so its total communication overhead for logistic regression is $2B \cdot t \cdot (1337\lambda + 8\ell + 1562)$ bits. As shown in Table 4, in the online phase our $\mathsf{S} - \mathsf{S}$ communication achieves $90.1 - 176\times$

|  | $n$ | Offline Comm (MB) |  | Online Comm (MB) |  |  | Total |
|---|---|---|---|---|---|---|---|
|  |  | S − S | C − S | S − S | C − S | Total | Comm |
| Our work | 10,240 | **0** | **0** | **2.85** | 266 | **269** | **269** |
|  | 100,352 | **0** | **0** | **28.0** | 2,605 | **2,633** | **2,633** |
| OT-based [29] | 10,240 | 24,151 | 0 | 257 | **123** | 380 | 24,531 |
|  | 100,352 | 236,607 | 0 | 2,524 | **1,204** | 3,728 | 240,335 |
| LHE-based [29] | 10,240 | 115 | 0 | 257 | **123** | 380 | 495 |
|  | 100,352 | 1,120* | 0 | 2,524 | **1,204** | 3,728 | 4,848* |
| Client-aided [29] | 10,240 | 0 | 614 | 502 | **123** | 625 | 1,239 |
|  | 100,352 | 0 | 6,016 | 4,424 | **1,204** | 5,628 | 11,644 |

Table 4: Communication cost of semi-honest logistic regression comparing our protocol to [29] instantiated with different offline approaches. "S − S" and "C − S" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

improvement over [29], and our total online communication achieves $1.41 - 2.32\times$ improvement. For the total communication (offline + online), we achieve $1.84 - 91.3\times$ improvement.

### 6.4 Neural Networks

We train a neural network consisting of three fully connected layers while the cross entropy function is employed as the loss function. The neural network has 128 neurons in each hidden layer and 10 in the output layers. We use the ReLU activation function for the two hidden layers and the MPC-friendly variant of the softmax function (see Section 3.3) for the output layer.

|  | $n$ | Offline Time (s) |  | Online Time (s) |  | Total Time (s) |  |
|---|---|---|---|---|---|---|---|
|  |  | LAN | WAN | LAN | WAN | LAN | WAN |
| Our work | 10,240 | **0** | **0** | 257 | **5875** | **257** | **5875** |
|  | 100,352 | **0** | **0** | 2,510* | **57,500*** | **2,510*** | **57,500*** |
| Client-aided [29] | 10,240 | 674 | 16,350* | **147** | 6,690* | 821 | 23,040* |
|  | 100,352 | 6,600* | 160,200* | **1,440*** | 65,600* | 8,040* | 225,800* |

Table 5: Running time of semi-honest neural networks on LAN and WAN networks comparing our protocol to client-aided [29]. * indicates estimated running time.

|  | $n$ | Offline Comm (MB) |  | Online Comm (MB) |  |  | Total |
|---|---|---|---|---|---|---|---|
|  |  | S − S | C − S | S − S | C − S | Total | Comm |
| Our work | 10,240 | **0** | **0** | **664** | 35,798 | **36,462** | **36,462** |
|  | 100,352 | **0** | **0** | **6,500*** | 351,000* | **357,500*** | **357,500*** |
| Client-aided [29] | 10,240 | 0 | 112,114 | 43,659 | **124** | 43,783 | 155,897 |
|  | 100,352 | 0 | 1,099,000* | 427,900* | **1,220*** | 429,100* | 1,528,000* |

Table 6: Communication cost of semi-honest neural networks comparing our protocol to client-aided [29]. "S − S" and "C − S" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

We compare the performance of our semi-honest neural network training to [29] in Tables 5 and 6. We only compare with the client-aided variant of [29] because it achieves the most comparable performance to ours. The neural network has two hidden layers with 128 neurons in each layer.

As shown in Table 5, in the online phase we achieve an improvement of 1.14× on WAN. For the total running time (offline + online), we achieve an improvement of 3.19× on LAN and 3.92× on WAN. As shown in Table 6, our $\mathsf{S}-\mathsf{S}$ communication in the online phase achieves an improvement of 65.8× and our total online communication achieves an improvement of 1.20×. We achieve a 4.28× improvement for the total communication (offline + online).

### 6.5 Security Against Malicious Clients

In this section, we report the performance of our protocols against malicious clients in Tables 7, 8, 9 for linear regression, logistic regression, and neural networks, respectively. When comparing with [29], we notice that OT-based and LHE-based variants are also secure against malicious clients because clients do not participate in the offline phase and only secret share their inputs in the online phase. We report the total running time on LAN and WAN networks for $n = 100,352, B = 128, d = 784, E = 2$.

| | Malicious Clients | Offline Time (s) | | Online Time (s) | | Total Time (s) | |
|---|---|---|---|---|---|---|---|
| | | LAN | WAN | LAN | WAN | LAN | WAN |
| Our malicious work | ✓ | 0 | 0 | 65.3 | 2,082 | 65.3 | 2,082 |
| Our semi-honest work | | 0 | 0 | 13.2 | 505 | 13.2 | 505 |
| OT-based [29] | ✓ | 2,667 | 36,600* | 25.8 | 557 | 2,692 | 37,157* |
| LHE-based [29] | ✓ | 13,800* | 14,000* | 25.8 | 557 | 13,826* | 14,557* |
| Client-aided [29] | | 52.0 | 749 | 35.3 | 1,126 | 87.3 | 1,875 |

Table 7: Running time of malicious linear regression on LAN and WAN networks comparing our protocol to [29] instantiated with different offline approaches. * indicates estimated running time.

For linear regression, the total time of our malicious protocol incurs a 4.95× overhead compared to our semi-honest protocol on LAN and a 4.12× overhead on WAN. This is consistent with our choice of $\delta = 4$ for verifying multiplication triples. Compared to OT-based and LHE-based variants of [29], we achieve an improvement of $41.2 - 212\times$ in the total time on LAN and $6.99 - 17.8\times$ in the total time on WAN.

| | Malicious Clients | Offline Time (s) | | Online Time (s) | | Total Time (s) | |
|---|---|---|---|---|---|---|---|
| | | LAN | WAN | LAN | WAN | LAN | WAN |
| Our malicious work | ✓ | 0 | 0 | 90.3 | 2,860 | 90.3 | 2,860 |
| Our semi-honest work | | 0 | 0 | 19.8 | 850 | 19.8 | 850 |
| OT-based [29] | ✓ | 2,667 | 36,600* | 40.0 | 1,056 | 2,707 | 37,656* |
| LHE-based [29] | ✓ | 13,800* | 14,000* | 40.0 | 1,056 | 13,840* | 15,056* |
| Client-aided [29] | | 55.9 | 941 | 46.3 | 1,398 | 102 | 2,339 |

Table 8: Running time of malicious logistic regression on LAN and WAN networks comparing our protocol to [29] instantiated with different offline approaches. * indicates estimated running time.

For logistic regression, the overhead of our malicious protocol compared to our semi-honest protocol is 4.56× on LAN and 3.36× on WAN. This is consistent with our choice of $\delta = 3$ for the activation function. Compared to OT-based and LHE-based variants of [29], we achieve an improvement of $30.0 - 153\times$ in the total time on LAN and $5.26 - 13.2\times$ in the total time on WAN.

| | Malicious Clients | Offline Time (s) | | Online Time (s) | | Total Time (s) | |
|---|---|---|---|---|---|---|---|
| | | LAN | WAN | LAN | WAN | LAN | WAN |
| Our malicious work | ✓ | 0 | 0 | 6,400* | 154,000* | 6,400* | 154,000* |
| Our semi-honest work | | 0 | 0 | 2,510* | 57,500* | 2,510* | 57,500* |
| Client-aided [29] | | 6,600* | 160,200* | 1,440* | 65,600* | 8,040* | 225,800* |

Table 9: Running time of malicious neural networks on LAN and WAN networks comparing our protocol to client-aided [29]. * indicates estimated running time.

For neural networks, the overhead of our malicious protocol compared to our semi-honest protocol is 2.55× on LAN and 2.68× on WAN. Compared to [29], we even outperform their semi-honest client-aided protocol (so we did not compare with the OT-based or LHE-based variant). In particular, we achieve an improvement of 1.26× on LAN and 1.49× on WAN in the total time. We expect the improvement to be even higher compared to their OT-based and LHE-based variants.

# 7 Acknowledgments

This project is supported in part by the NSF CNS Award 2247352, Brown Data Science Seed Grant, Meta Research Award, Google Research Scholar Award, and Amazon Research Award.

# References

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In *SCN*, 2022.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[3] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In *ACM SIGSAC CCS*, 2020.

[4] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. *CoRR*, 2016.

[5] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *CCS*, 2007.

[6] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: fast and robust framework for privacy-preserving machine learning. *Proc. Priv. Enhancing Technol.*, 2020.

[7] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[8] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *ACM SIGSAC*, 2019.

[9] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*, 2020.

[10] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *USENIX NSDI*, 2017.

[11] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 2012.

[12] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, 2017.

[13] Jiahui Geng, Yongli Mou, Feifei Li, Qing Li, Oya Beyan, Stefan Decker, and Chunming Rong. Towards general deep leakage in federated learning. *CoRR*, 2021.

[14] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[15] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *ACM SIGKDD*, 2005.

[16] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[17] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *CoRR*, 2019.

[18] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-element sets. In *FC*, 2014.

[19] Jakub Konečný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, 2016.

[20] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *USENIX Security*, 2021.

[21] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *IEEE SP*, 2020.

[22] Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Two-party private set intersection with an untrusted third party. In *SIGSAC*, 2019.

[23] Hsiao-Ying Lin and Wen-Guey Tzeng. An efficient solution to the millionaires' problem based on homomorphic encryption. In *ACNS*, 2005.

[24] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *J. Cryptol.*, 2002.

[25] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[26] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE SP*, 2019.

[27] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, 2020.

[28] Payman Mohassel and Peter Rindal. Aby$^3$: A mixed protocol framework for machine learning. In *ACM SIGSAC CCS*, 2018.

[29] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE SP*, 2017.

[30] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In *USENIX Security*, 2021.

[31] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

[32] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *ACM SIGSAC CCS*, 2020.

[33] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*, 2018.

[34] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. In *USENIX Security*, 2020.

[35] Jaideep Vaidya, Hwanjo Yu, and Xiaoqian Jiang. Privacy-preserving SVM classification. *Knowl. Inf. Syst.*, 2008.

[36] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019.

[37] Zhibo Wang, Mengkai Song, Zhifei Zhang, Yang Song, Qian Wang, and Hairong Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *IEEE INFOCOM*, 2019.

[38] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[39] Hwanjo Yu, Jaideep Vaidya, and Xiaoqian Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *PAKDD*, 2006.

[40] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In *NeurIPS*, 2019.

# A Security Against Malicious Clients

In this section, we enhance the security of our protocols to protect against malicious clients. At a high level, we only need to verify that the client-aided inner product and the client-aided sign check are computed correctly, which we present in Sections A.1 and A.2 respectively. The client-aided ReLU, client-aided division, and linear/logistic regression only rely on these two building blocks. For neural networks, we additionally need to guarantee that the Beaver multiplication triples are generated correctly, which we show in Section A.3. Finally, we present the performance evaluation in Section 6.5

## A.1 Client-Aided Inner Product

We present a general protocol for verifying the correctness of matrix multiplication for any dimension (instead of only for vectors). This allows for optimizations that we discuss at the end of this section.

**Construction Overview.** First, we present a subprotocol $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ to verify whether a secret shared triple $(\mathbf{X} \in \mathbb{Z}_{2^\ell}^{a\times b}, \mathbf{Y} \in \mathbb{Z}_{2^\ell}^{b\times c}, \mathbf{Z} \in \mathbb{Z}_{2^\ell}^{a\times c})$ is correct by revealing the triple in the clear. Here one triple $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ is correct means $\mathbf{X} \cdot \mathbf{Y} = \mathbf{Z}$. In this subprotocol, each server $\mathsf{S}_i$ has the secret sharing of the triple $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y} \rrbracket_i, \llbracket \mathbf{Z} \rrbracket_i)$ as input. Since the servers are both semi-honest, we simply let each server send the secret shares to each other, reconstruct the triple, and check whether $\mathbf{X} \cdot \mathbf{Y} = \mathbf{Z}$. The communication cost between the two servers is $2(a \cdot b + b \cdot c + a \cdot c)$ ring elements in total. The subprotocol is described in Figure 13.

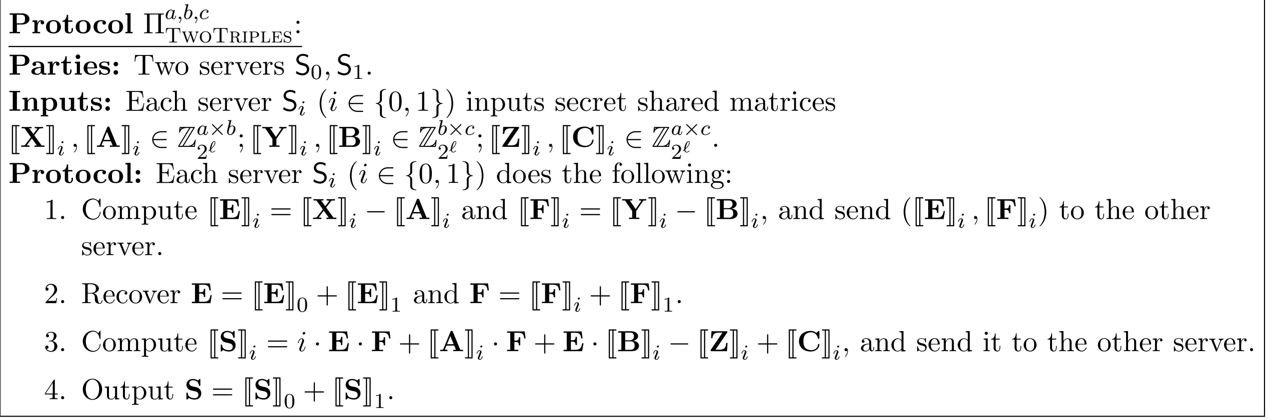
**Protocol $\Pi^{a,b,c}_{\textsc{OpenTriple}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$.
**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs secret shared matrices
$\llbracket \mathbf{X} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times b}, \llbracket \mathbf{Y} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{b\times c}, \llbracket \mathbf{Z} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times c}$.
**Protocol:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ does the following:

1. Send $\llbracket \mathbf{X} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times b}, \llbracket \mathbf{Y} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{b\times c}, \llbracket \mathbf{Z} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times c}$ to the other server.
2. Recover $\mathbf{X} = \llbracket \mathbf{X} \rrbracket_0 + \llbracket \mathbf{X} \rrbracket_1$, $\mathbf{Y} = \llbracket \mathbf{Y} \rrbracket_0 + \llbracket \mathbf{Y} \rrbracket_1$, $\mathbf{Z} = \llbracket \mathbf{Z} \rrbracket_0 + \llbracket \mathbf{Z} \rrbracket_1$.
3. Outputs 1 if $\mathbf{X} \cdot \mathbf{Y} = \mathbf{Z}$ and 0 otherwise.

Figure 13: Subprotocol $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ for verifying a multiplication triple by revealing the matrices in the clear. Each server outputs 1 if $\mathbf{X} \cdot \mathbf{Y} = \mathbf{Z}$ and 0 otherwise.

We present another subprotocol $\Pi^{a,b,c}_{\textsc{TwoTriples}}$ to verify one multiplication triple $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ using another multiplication triple $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ without opening. Through the protocol, the servers can get the result of $(\mathbf{X} \cdot \mathbf{Y} - \mathbf{Z}) - (\mathbf{A} \cdot \mathbf{B} - \mathbf{C})$. Then the servers can gain some information from this result, in particular, if the result is non-zero, then the two triples cannot be both correct. In

this subprotocol, each server $\mathsf{S}_i$ has the secret sharings of the two triples $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y} \rrbracket_i, \llbracket \mathbf{Z} \rrbracket_i)$ and $(\llbracket \mathbf{A} \rrbracket_i, \llbracket \mathbf{B} \rrbracket_i, \llbracket \mathbf{C} \rrbracket_i)$ as input. Then the servers reconstruct $\mathbf{E} = \mathbf{X} - \mathbf{A}$ and $\mathbf{F} = \mathbf{Y} - \mathbf{B}$ and use these to recover and output $\mathbf{A} \cdot \mathbf{F} + \mathbf{E} \cdot \mathbf{B} + \mathbf{E} \cdot \mathbf{F} - \mathbf{Z} + \mathbf{C}$, which is equal to $(\mathbf{X} \cdot \mathbf{Y} - \mathbf{Z}) - (\mathbf{A} \cdot \mathbf{B} - \mathbf{C})$. The communication cost between the two servers is $2(a \cdot b + b \cdot c + a \cdot c)$ ring elements in total. The subprotocol is described in Figure 14.


**Protocol $\Pi^{a,b,c}_{\text{TwoTriples}}$:**

**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$.

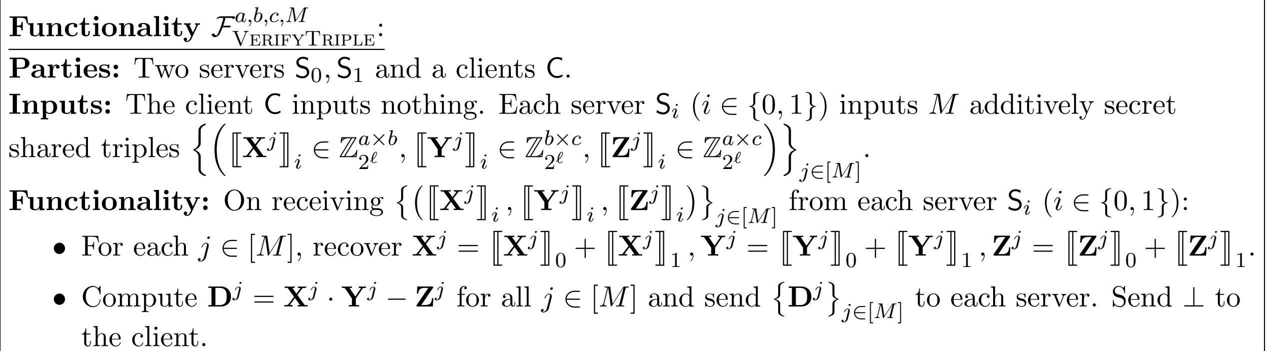
**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs secret shared matrices $\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{A} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a \times b}; \llbracket \mathbf{Y} \rrbracket_i, \llbracket \mathbf{B} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{b \times c}; \llbracket \mathbf{Z} \rrbracket_i, \llbracket \mathbf{C} \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a \times c}$.

**Protocol:** Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ does the following:

1. Compute $\llbracket \mathbf{E} \rrbracket_i = \llbracket \mathbf{X} \rrbracket_i - \llbracket \mathbf{A} \rrbracket_i$ and $\llbracket \mathbf{F} \rrbracket_i = \llbracket \mathbf{Y} \rrbracket_i - \llbracket \mathbf{B} \rrbracket_i$, and send $(\llbracket \mathbf{E} \rrbracket_i, \llbracket \mathbf{F} \rrbracket_i)$ to the other server.
2. Recover $\mathbf{E} = \llbracket \mathbf{E} \rrbracket_0 + \llbracket \mathbf{E} \rrbracket_1$ and $\mathbf{F} = \llbracket \mathbf{F} \rrbracket_i + \llbracket \mathbf{F} \rrbracket_1$.
3. Compute $\llbracket \mathbf{S} \rrbracket_i = i \cdot \mathbf{E} \cdot \mathbf{F} + \llbracket \mathbf{A} \rrbracket_i \cdot \mathbf{F} + \mathbf{E} \cdot \llbracket \mathbf{B} \rrbracket_i - \llbracket \mathbf{Z} \rrbracket_i + \llbracket \mathbf{C} \rrbracket_i$, and send it to the other server.
4. Output $\mathbf{S} = \llbracket \mathbf{S} \rrbracket_0 + \llbracket \mathbf{S} \rrbracket_1$.

Figure 14: Subprotocol $\Pi^{a,b,c}_{\text{TwoTriples}}$ for verifying one multiplication triple using another multiplication triple without opening. Each server outputs $(\mathbf{X} \cdot \mathbf{Y} - \mathbf{Z}) - (\mathbf{A} \cdot \mathbf{B} - \mathbf{C})$.

With these two subprotocols, we use cut-and-choose to design a protocol $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$ to verify $M$ multiplication triples together. The ideal functionality is described in Figure 15.


**Functionality $\mathcal{F}^{a,b,c,M}_{\text{VerifyTriple}}$:**

**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a clients $\mathsf{C}$.

**Inputs:** The client $\mathsf{C}$ inputs nothing. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ inputs $M$ additively secret shared triples $\left\{ \left( \llbracket \mathbf{X}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a \times b}, \llbracket \mathbf{Y}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{b \times c}, \llbracket \mathbf{Z}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a \times c} \right) \right\}_{j \in [M]}$.

**Functionality:** On receiving $\{ ( \llbracket \mathbf{X}^j \rrbracket_i, \llbracket \mathbf{Y}^j \rrbracket_i, \llbracket \mathbf{Z}^j \rrbracket_i ) \}_{j \in [M]}$ from each server $\mathsf{S}_i$ $(i \in \{0,1\})$:

- For each $j \in [M]$, recover $\mathbf{X}^j = \llbracket \mathbf{X}^j \rrbracket_0 + \llbracket \mathbf{X}^j \rrbracket_1, \mathbf{Y}^j = \llbracket \mathbf{Y}^j \rrbracket_0 + \llbracket \mathbf{Y}^j \rrbracket_1, \mathbf{Z}^j = \llbracket \mathbf{Z}^j \rrbracket_0 + \llbracket \mathbf{Z}^j \rrbracket_1$.
- Compute $\mathbf{D}^j = \mathbf{X}^j \cdot \mathbf{Y}^j - \mathbf{Z}^j$ for all $j \in [M]$ and send $\{\mathbf{D}^j\}_{j \in [M]}$ to each server. Send $\perp$ to the client.

Figure 15: Ideal functionality $\mathcal{F}^{a,b,c,M}_{\text{VerifyTriple}}$ for verifying $M$ multiplication triples that are secret shared among the servers.

Intuitively, the client $\mathsf{C}$ generates $N = \delta \cdot M + \mu$ new multiplication triples. To ensure these triples are generated correctly, the servers first randomly pick $\mu$ triples to open (in Step 4). Once these checks are passed, the servers randomly partition the remaining triples into $M$ groups, each of size $\delta$, and each group is used to verify one triple. The protocol is described in Figure 16.

**Theorem A.1.** *For the parameters we choose below, the protocol $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$ (Figure 16) securely computes the ideal functionality $\mathcal{F}^{a,b,c,M}_{\text{VerifyTriple}}$ (Figure 15) against an adversary that corrupts either one of the two servers $\mathsf{S}_0, \mathsf{S}_1$ in a semi-honest way, or the client $\mathsf{C}$ maliciously.*

**Proof Sketch.** For security against a semi-honest server $\mathsf{S}_0$ (the proof for a semi-honest $\mathsf{S}_1$ is almost identical), note that $\mathsf{S}_0$ receives random secret shares in Step 1c, verifies the randomly

**Protocol $\Pi^{a,b,c,M}_{\textsc{VerifyTriple}}$:**

0. The two servers $\mathsf{S}_0$ and $\mathsf{S}_1$ and the client $\mathsf{C}$ agree on parameters $\delta, \mu$. Let $N := \delta \cdot M + \mu$.
1. The client $\mathsf{C}$ does the following:
   (a) For each $k \in [N]$, sample random matrices $\mathbf{A}^k \xleftarrow{\$} \mathbb{Z}^{a\times b}_{2^\ell}, \mathbf{B}^k \xleftarrow{\$} \mathbb{Z}^{b\times c}_{2^\ell}$ and compute $\mathbf{C}^k = \mathbf{A}^k \cdot \mathbf{B}^k$.
   (b) For each $k \in [N]$, generate additive secret sharings $(\llbracket \mathbf{A}^k \rrbracket_0, \llbracket \mathbf{A}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{A}^k)$, $(\llbracket \mathbf{B}^k \rrbracket_0, \llbracket \mathbf{B}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{B}^k), (\llbracket \mathbf{C}^k \rrbracket_0, \llbracket \mathbf{C}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{C}^k)$.
   (c) Send $\{(\llbracket \mathbf{A}^k \rrbracket_i, \llbracket \mathbf{B}^k \rrbracket_i, \llbracket \mathbf{C}^k \rrbracket_i)\}_{k\in[N]}$ to each server $\mathsf{S}_i$ $(i \in \{0,1\})$.
2. Server $\mathsf{S}_0$ randomly samples a permutation $\pi : [N] \to [N]$ and sends it to $\mathsf{S}_1$.
3. Each server $\mathsf{S}_i$ $(i \in \{0,1\})$ uses $\pi$ to shuffle its triples and obtain $\left\{\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)\right\}_{k\in[N]}$. In particular, $\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right) := (\llbracket \mathbf{A}^{\pi(k)} \rrbracket_i, \llbracket \mathbf{B}^{\pi(k)} \rrbracket_i, \llbracket \mathbf{C}^{\pi(k)} \rrbracket_i)$.
4. The two servers run $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ on each of the first $\mu$ shuffled triples, namely $\left\{\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)\right\}_{k=1}^{\mu}$. The servers abort the protocol if any $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ instance outputs 0.
5. For each $j \in [M]$:
   (a) For each $t \in [\delta]$, let $k := \mu + \delta \cdot (j-1) + t$, and let the two servers run $\Pi^{a,b,c}_{\textsc{TwoTriples}}$ on $(\llbracket \mathbf{X}^j \rrbracket_i, \llbracket \mathbf{Y}^j \rrbracket_i, \llbracket \mathbf{Z}^j \rrbracket_i)$ and $\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)$ to learn $\mathbf{D}^j_t$.
   (b) If $\mathbf{D}^j_t$ is the same for all $t \in [\delta]$, then let $\mathbf{D}^j := \mathbf{D}^j_1$; otherwise abort the protocol.
6. Both servers output $\{\mathbf{D}^j\}_{j\in[M]}$.

Figure 16: Protocol $\Pi^{a,b,c,M}_{\textsc{VerifyTriple}}$ for verifying $M$ multiplication triples that are secret shared among the servers.

generated multiplication triples in Step 4, and learns $\mathbf{D}^j$ for each $j \in [M]$ in Step 5. Simulation involves generating random shares in Step 1c, revealing consistent shares in $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ in Step 4, and revealing $\mathbf{D}^j$ (from the ideal functionality) in $\Pi^{a,b,c}_{\textsc{TwoTriples}}$ in Step 5. Note that in $\Pi^{a,b,c}_{\textsc{TwoTriples}}$, the recovered $\mathbf{E}, \mathbf{F}$ are both random matrices and do not reveal any information. The correctness of the protocol is also easy to verify.

For security against an adversary $\mathcal{A}$ that corrupts the client $\mathsf{C}$ maliciously, note that it only sends randomly generated shares in Step 1c. We construct a simulator that on receiving the triples from $\mathcal{A}$, verifies the correctness of all the triples, namely, $\mathbf{A}^k \cdot \mathbf{B}^k = \mathbf{C}^k$ for all $k \in [M]$. If all these triples are valid, then the simulator tells the ideal functionality to continue; otherwise, it tells the ideal functionality to abort. The only difference between the real-world execution and the simulation is when the malicious client succeeds in cheating, which happens with negligible probability for the parameters we choose below.

**Parameters.** As discussed in the protocol, we use $N = \delta \cdot M + \mu$ newly generated triples to check $M$ triples. We need to calculate the probability that the malicious client succeeds in generating

incorrect triples without being caught, and we would like to make this probability negligible in the statistical security parameter $\sigma$. In particular, our goal is to choose appropriate parameters $M, \delta, \mu$ such that the probability is no more than $2^{-\sigma}$.

We consider a malicious client $\mathcal{A}$'s best strategy and bound the success probability. We start by defining the following balls-and-buckets game for the newly generated triples.

$\mathsf{Game}(\mathcal{A}, M, \delta, \mu)$ :

1. The adversary $\mathcal{A}$ prepares $N = \delta \cdot M + \mu$ balls. Each ball can be either *good* or *bad*.
2. $\mu$ random balls are chosen and opened. If one of the $\mu$ balls is bad, then output 0. Otherwise, the game proceeds to the next step.
3. The remaining $\delta \cdot M$ balls are randomly divided into $M$ buckets of equal size $\delta$. We say that a bucket is fully good if all balls inside it are good. Similarly, a bucket is fully bad if all balls inside it are bad.
4. The output of the game is 1 if and only if each bucket is either fully good or fully bad, and there exists at least one fully bad bucket.

$\mathsf{Game}'(\mathcal{A}, M, \delta, \mu)$ :

1. The adversary $\mathcal{A}$ prepares $M$ balls to be checked and $N = \delta \cdot M + \mu$ new balls. Each ball can be either *good* or *bad*.
2. $\mu$ random balls among $N$ new balls are chosen and opened. If one of the $\mu$ balls is bad, then output 0. Otherwise, the game proceeds to the next step.
3. The remaining $\delta \cdot M$ balls are randomly divided into $M$ buckets of equal size $\delta$. The $M$ balls to be checked are also randomly assigned to these $M$ buckets, one per bucket. We say that a bucket is fully good if all balls inside it are good. Similarly, a bucket is fully bad if all balls inside it are bad.
4. The output of the game is 1 if and only if each bucket is either fully good or fully bad, and there exists at least one fully bad bucket.

Note that the output condition in the last step of $\mathsf{Game}$ and $\mathsf{Game}'$ enforces the adversary to choose at least one bad ball if it wishes to win. We first observe that for $\mathcal{A}$ to win the game, the number of bad balls $\mathcal{A}$ chooses must be a multiple of $M$, the size of a bucket.

**Lemma A.2.** *Let $T$ be the number of bad balls chosen by the adversary $\mathcal{A}$. Then a necessary condition for $\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1$ is that $T = \delta \cdot t$ for some $t \in [M]$.*

Let $t$ be the number of buckets $\mathcal{A}$ has chosen to corrupt. Then for any $0 < t \leq B$, it holds that

$$\Pr\left[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1\right] = \binom{M}{t}\binom{\delta M + \mu}{t\delta}^{-1}.$$

$$\begin{aligned}\Pr[\mathsf{Game}'(\mathcal{A}, M, \delta, \mu) = 1] &= \Pr[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1] \cdot \binom{M}{t}^{-1} \\ &= \binom{\delta \cdot M + \mu}{t\delta}^{-1}.\end{aligned}$$

If $\mu \leq \delta$, the best strategy of $\mathcal{A}$ is to corrupt $M$ buckets. In this case, when $M$ and $\delta$ remain the same, the upper bounds of $\Pr[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1]$ and $\Pr[\mathsf{Game}'(\mathcal{A}, M, \delta, \mu) = 1]$ increase exponentially as $\mu$ decreases. However, increasing $\delta$ is more costly than increasing $\mu$, concerning both communication and computation overhead. From the honest parties' perspective, the optimal strategy is to set a greater $\mu$ so that $\delta$ can be smaller when the adversary's success probability is fixed. Hence we will assume $\mu \geq \delta$.

Given that $\mu \geq \delta$, the best strategy of $\mathcal{A}$ is to corrupt exactly one bucket. This allows us to derive an upper bound of the success probability of the adversary.

**Theorem A.3.** *If $\mu \geq \delta$, then for any adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\mathsf{Game}'(\mathcal{A}, M, \delta, \mu) = 1\right] \leq \binom{\delta \cdot M + \mu}{\delta}^{-1}.$$

*Proof.* Since $\mu \geq \delta$, we can easily see that

$$\begin{aligned}
\Pr\left[\mathsf{Game}'(\mathcal{A}, M, \delta, \mu) = 1\right] &\leq \min_{t \geq 1} \left\{ \binom{\delta \cdot M + \mu}{t\delta}^{-1} \right\} \\
&= \min \left\{ \binom{\delta \cdot M + \mu}{\delta}^{-1}, \binom{\delta \cdot M + \mu}{\delta \cdot M}^{-1} \right\} \\
&= \min \left\{ \binom{\delta \cdot M + \mu}{\delta}^{-1}, \binom{\delta \cdot M + \mu}{\mu}^{-1} \right\} \\
&= \binom{\delta \cdot M + \mu}{\delta}^{-1}.
\end{aligned}$$

□

Therefore, to guarantee that $\Pr\left[\mathsf{Game}'(\mathcal{A}, M, \delta, \mu) = 1\right] \leq 2^{-\sigma}$, we only need to guarantee that $\binom{\delta \cdot M+\mu}{\delta}^{-1} \leq 2^{-\sigma}$, which is equivalent to $\binom{\delta \cdot M+\mu}{\delta} \geq 2^{\sigma}$.

From the above formula, we observe that the change of $\mu$ has little influence on the adversary's success probability when $\mu \geq \delta$. Therefore, we set $\mu := \delta$ to minimize $\frac{\delta \cdot M+\mu}{M}$ and achieve a minimum communication overhead. In Table 10, we present the minimum $M$ (power of 2) for $\sigma = 40$ and different $\delta$ and $\mu$ values.

| $\delta$ | $\mu$ | $M$ | $N = \delta \cdot M + \mu$ |
|---|---|---|---|
| 6 | 6 | 128($2^7$) | 774 |
| 5 | 5 | 256($2^8$) | 1,285 |
| 4 | 4 | 1,024($2^{10}$) | 4,100 |
| 3 | 3 | 8,192($2^{13}$) | 24,579 |
| 2 | 2 | 1,048,576($2^{20}$) | 1,048,578 |

Table 10: Parameters for verifying multiplication triples and sign check results for $\sigma = 40$.

**Communication and Optimizations.** We can use PRF to reduce communication similarly as discussed earlier. In Steps 1b and 1c, the client can use the shared PRF key with one server $\mathsf{S}_0$ to

generate $\left(\llbracket \mathbf{A}^k \rrbracket_0, \llbracket \mathbf{B}^k \rrbracket_0, \llbracket \mathbf{C}^k \rrbracket_0\right)$ without communication, and send the other share $\left(\llbracket \mathbf{A}^k \rrbracket_1, \llbracket \mathbf{B}^k \rrbracket_1, \llbracket \mathbf{C}^k \rrbracket_1\right)$ to $\mathsf{S}_1$. This can reduce the communication between the client and the servers to $N \cdot (a \cdot c)$ ring elements. Besides, in Step 2, the two servers can share a PRF key with each other and then use the PRF key to generate a random permutation $\pi : [N] \to [N]$ without communication.

Moreover, in Step 5, we let the servers only verify whether $\mathbf{D}_t^j$ is equal to $\mathbf{0}$ for all $j \in [M], t \in [\delta]$ and use linear combinations and PRF to reduce communication in our implementation. To be more specific, we first let the servers use the shared PRF key to randomly sample $\delta \cdot M$ shared values $\eta_1, \ldots, \eta_{\delta \cdot M} \in \mathbb{Z}_{2^\ell}$ without communication, then each server $\mathsf{S}_i$ uses these as $\delta \cdot M$ coefficients to construct a linear combination of $\llbracket \mathbf{D}_t^j \rrbracket_i$ for all $j \in [M], t \in [\delta]$. Finally, each server $\mathsf{S}_i$ sends $\sum_{j=1}^{M} \sum_{t=1}^{\delta} \eta_{\delta \cdot (j-1)+t} \cdot \llbracket \mathbf{D}_t^j \rrbracket_i$ to the other server and checks whether the sum is equal to $\mathbf{0}$. The probability of failure is negligible. This can reduce the communication of Step 5 between the two servers to $2(\delta \cdot M) \cdot (a \cdot b + b \cdot c) + 2(a \cdot c)$ ring elements.

Hence the final communication between the two servers is $2(\delta \cdot M + \mu) \cdot (a \cdot b + b \cdot c) + 2(\mu + 1) \cdot (a \cdot c)$ ring elements and the communication between the client and the servers is $(\delta \cdot M + \mu) \cdot (a \cdot c)$ ring elements.

In our implementation, we further improve communication and storage when the verification protocol is applied to linear regression, logistic regression, and neural networks. In one iteration of semi-honest linear regression and logistic regression, each client in the mini-batch generates two multiplication triples. With the optimizations we mentioned in Section 5.1, the clients' data $\mathbf{x}$ is used twice in two multiplication triples and all the clients use the same mask of $\mathbf{w}$. We can apply similar optimizations during the verification phase. In particular, we pack these $2B$ triples into a *large triple* in which these $2B$ triples are combined into two matrix multiplications, namely $\mathbf{X} \times \mathbf{w}_{\mathsf{mask}}$ and $\mathbf{X}^\top \times \mathbf{Y}'_{\mathsf{mask}}$, where $\mathbf{X}$ denotes the matrix of clients' data, $\mathbf{w}_{\mathsf{mask}}$ and $\mathbf{Y}'_{\mathsf{mask}}$ denote the mask of $\mathbf{w}$ and the mask of $\mathbf{X} \times \mathbf{w} - \mathbf{Y}$. In the large triple, all the reused values are stored only once. We can directly verify the large triple instead of verifying $2B$ triples separately. More specifically, we let the clients generate new large triples with the same dimension to verify the previously computed large triples. In this case, we only have one large triple to be verified in each iteration, but we improve approximately $4\times$ of the communication in the verification phase. The communication among the servers is $2(\delta \cdot M + \mu) \cdot (Bd + B + d) + 2(\mu + 1) \cdot (B + d)$ ring elements and the communication between the client and the servers is $(\delta \cdot M + \mu) \cdot (B + d)$ ring elements. Similarly, in each layer of neural networks, we also pack all the multiplication triples into one *large triple*.

Taking the servers' storage cost into consideration, we do not expect $M$ and $N$ to be too large. We also need to consider the total number of the multiplication triples that need to verify since M shouldn't exceed this. In our experiment for linear regression, logistic regression, and neural networks with $n = 100,352$, we pick $M = 1024, \delta = 4, \mu = 4$, which means we call the protocol $\Pi_{\textsc{VerifyTriple}}^{a,b,c,M}$ every 1024 iterations.

## A.2 Client-Aided Sign Check

**Construction Overview.** In this section, we use similar ideas to verify sign check results. First, For $b^{\mathsf{S}} \in \{0, 1\}$ and $b^{\mathsf{C}} \in \mathbb{Z}_{2^\ell}$, we define a function

$$\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) := b^{\mathsf{S}} + (1 - 2b^{\mathsf{S}}) \cdot b^{\mathsf{C}}.$$

Note that $\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) = b^{\mathsf{C}}$ if $b^{\mathsf{S}} = 0$ and $\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) = 1 - b^{\mathsf{C}}$ if $b^{\mathsf{S}} = 1$. In other words, $\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) = b^{\mathsf{S}} \oplus b^{\mathsf{C}}$ if $b^{\mathsf{C}} \in \{0, 1\}$.

Similar to client-aided inner product, we present a protocol $\Pi_{\textsc{TwoSignCheck}}$ to verify one sign check result $(b^{\mathsf{S}}, b^{\mathsf{C}})$ using another sign check result $(c^{\mathsf{S}}, c^{\mathsf{C}})$ without opening. From the protocol, the servers can get the value of $b^{\mathsf{S}} \oplus b^{\mathsf{C}} - c^{\mathsf{S}} \oplus c^{\mathsf{C}} = \mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) - \mathsf{Diff}(c^{\mathsf{S}}, c^{\mathsf{C}})$. Then the servers can gain some information from it, in particular, if the result is non-zero, then the two sign check results cannot be both correct. In this protocol, each server $\mathsf{S}_i$ has the input of two sign check results: $(b^{\mathsf{S}}, [\![b^{\mathsf{C}}]\!]_i)$ and $(c^{\mathsf{S}}, [\![c^{\mathsf{C}}]\!]_i)$. Then the servers reconstruct and output $b^{\mathsf{S}} - c^{\mathsf{S}} + (1 - 2b^{\mathsf{S}}) \cdot b^{\mathsf{C}} - (1 - 2c^{\mathsf{S}}) \cdot c^{\mathsf{C}}$, which is equal to $\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) - \mathsf{Diff}(c^{\mathsf{S}}, c^{\mathsf{C}})$. The communication is 2 ring elements between the servers. The protocol is described in Figure 17.

**Protocol $\Pi_{\textsc{TwoSignCheck}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$.
**Inputs:** Each server $\mathsf{S}_i$ $(i \in \{0, 1\})$ inputs $b^{\mathsf{S}}, c^{\mathsf{S}} \in \{0, 1\}$ and additively secret shared values $[\![b^{\mathsf{C}}]\!]_i, [\![c^{\mathsf{C}}]\!]_i \in \mathbb{Z}_{2^\ell}$.
**Protocol:** Each server $\mathsf{S}_i$ $(i \in \{0, 1\})$ does the following:

1. Compute $[\![e]\!]_i = i \cdot (b^{\mathsf{S}} - c^{\mathsf{S}}) + (1 - 2b^{\mathsf{S}}) \cdot [\![b^{\mathsf{C}}]\!]_i - (1 - 2c^{\mathsf{S}}) \cdot [\![c^{\mathsf{C}}]\!]_i$ and send it to the other server.
2. Output $e = [\![e]\!]_0 + [\![e]\!]_1$.

Figure 17: Protocol $\Pi_{\textsc{TwoSignCheck}}$ for verifying one sign check result using another result. Each server outputs $\mathsf{Diff}(b^{\mathsf{S}}, b^{\mathsf{C}}) - \mathsf{Diff}(c^{\mathsf{S}}, c^{\mathsf{C}})$.

Given the subprotocol $\Pi_{\textsc{TwoSignCheck}}$, we use cut-and-choose to design a protocol $\Pi^M_{\textsc{VerifySignCheck}}$ to verify $M$ sign check results together. The ideal functionality is described in Figure 18.

**Functionality $\mathcal{F}^M_{\textsc{VerifySignCheck}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a clients $\mathsf{C}$.
**Inputs:** The client $\mathsf{C}$ inputs nothing. Each server $\mathsf{S}_i$ $(i \in \{0, 1\})$ inputs $M$ tuples $\{([\![x^j]\!]_i \in \mathbb{Z}_{2^\ell}, b^{j,\mathsf{S}} \in \{0, 1\}, [\![b^{j,\mathsf{C}}]\!]_i \in \mathbb{Z}_{2^\ell})\}_{j \in [M]}$.
**Functionality:** On receiving $\{([\![x^j]\!]_i, b^{j,\mathsf{S}}, [\![b^{j,\mathsf{C}}]\!]_i)\}_{j \in [M]}$ from each server $\mathsf{S}_i$ $(i \in \{0, 1\})$:

- For each $j \in [M]$, recover $x^j = [\![x^j]\!]_0 + [\![x^j]\!]_1, b^{j,\mathsf{C}} = [\![b^{j,\mathsf{C}}]\!]_0 + [\![b^{j,\mathsf{C}}]\!]_1$.
- If $\exists j \in [M]$ where $b^{j,\mathsf{C}} \neq 0$ or 1, or $b^{j,\mathsf{S}} \oplus b^{j,\mathsf{C}} \neq (x^j > 0)$, then send $(\mathsf{error}, j)$ along with $e := \mathsf{Diff}(b^{j,\mathsf{S}}, b^{j,\mathsf{C}}) - (x^j > 0)$ to each server and halt.
- Send $\mathsf{correct}$ to each server.

Figure 18: Ideal functionality $\mathcal{F}^M_{\textsc{VerifySignCheck}}$ for verifying $M$ sign check results with the assistance of a client.

Intuitively, for each input $([\![x]\!]_i, b^{\mathsf{S}}, [\![b^{\mathsf{C}}]\!]_i)$ that needs to be verified, the servers first generate $\delta$ equivalent instances to check. Specifically, for each instance the servers sample a random value $d \in \mathbb{Z}_{2^\ell}$, then $\mathsf{S}_0$ computes $[\![x]\!]_0 + d$ and $\mathsf{S}_1$ computes $[\![x]\!]_1 - d$, which becomes a fresh secret sharing of $x$. Then they perform a client-aided sign check on this new secret sharing. Moreover, the servers randomly sample $\mu$ new values, for which they know the sign check result, and then perform the sign check with the client.

The servers randomly shuffle all these $N = \delta \cdot M + \mu$ instances and run sign check with the client. Afterwards, for the $\mu$ new values, the servers can simply verify if their results are correct by opening the results. Once these checks are passed, we can think of it as the servers randomly partitioning the remaining triples into $M$ groups, each of size $\delta$. They run the protocol $\Pi_{\text{TwoSignCheck}}$ to check each sign check result using $\delta$ newly computed results. The protocol is described in Figure 19.

**Theorem A.4.** *For the parameters we choose below, the protocol $\Pi^M_{\text{VerifySignCheck}}$ (Figure 19) securely computes the ideal functionality $\mathcal{F}^M_{\text{VerifySignCheck}}$ (Figure 18) against an adversary that corrupts either one of the two servers $\mathsf{S}_0, \mathsf{S}_1$ in a semi-honest way, or the client $\mathsf{C}$ maliciously.*

**Proof Sketch.** For security against a semi-honest server $\mathsf{S}_0$ (the proof for a semi-honest $\mathsf{S}_1$ is almost identical), note that in Step 5 the two servers perform $\Pi_{\text{SignCheck}}$ on their randomly sample values and compare with their pre-computed output. Simulation for this step only needs to generate a random $[\![c^{k,\mathsf{C}}]\!]_0$ in $\Pi_{\text{SignCheck}}$ and reveal $e = 0$ in $\Pi_{\text{TwoSignCheck}}$. In Step 6 the two servers perform $\Pi_{\text{SignCheck}}$ on $x^j$ and compare with their pre-computed output. To simulate this step, the simulator first generates a random $[\![c^{k,\mathsf{C}}]\!]_0$ in $\Pi_{\text{SignCheck}}$. If the output from the ideal functionality is $(\mathsf{error}, j)$ for some $j$ along with some $e$, then the simulator reveals this $e$ in $\Pi_{\text{TwoSignCheck}}$ for this $j$ (on all $t \in [\delta]$); otherwise it reveals $e = 0$. The simulated view is indistinguishable from the real-world execution because $\mathsf{S}_0$ only receives a random share of $b^{\mathsf{C}}$ from the client in $\Pi_{\text{SignCheck}}$, and $\Pi_{\text{TwoSignCheck}}$ reveals nothing beyond the output $e$. Correctness follows naturally from the cross check done in $\Pi_{\text{TwoSignCheck}}$.

For security against an adversary $\mathcal{A}$ that corrupts the client $\mathsf{C}$ maliciously, note that it only sends randomly generated shares for $c^{k,\mathsf{C}}$ in $\Pi_{\text{SignCheck}}$ (in Steps 5 and 6). We construct a simulator that emulates the simulator for a semi-honest client in $\Pi_{\text{SignCheck}}$. On receiving the shares from $\mathcal{A}$, it verifies the correctness of the shares, namely, $[\![c^{k,\mathsf{C}}]\!]_0 + [\![c^{k,\mathsf{C}}]\!]_1$ is correct. If all these shares are correct, then the simulator tells the ideal functionality to continue; otherwise it tells the ideal functionality to abort. The only difference between the real-world execution and the simulation is when the malicious client succeeds in cheating, which happens with negligible probability for the parameters we choose below.

**Parameters.** The analysis for the probability that the malicious client succeeds in cheating is exactly the same as in Section A.1. We refer to Table 10 for the choice of parameters.

**Communication and Optimizations.** Similarly as discussed earlier, we can use PRF to reduce the communication in Step 3 and use linear combinations to reduce the communication in Step 6. In Steps 1a and 2a, we can let the two servers share a PRF key at the beginning, then use the PRF key to generate a (pseudo)random value $d$ without communication. In Step 6, we combine $e^j_t - e^j_1$ ($t \in \{2, \ldots, \delta\}$) for all $j \in [m]$ by using linear combination and open the result to check whether it is 0, reducing the communication from $2M \cdot \delta$ ring elements to 2 ring elements. Thus the communication of the protocol $\Pi^M_{\text{VerifySignCheck}}$ between the two servers is $2\mu + 2$ ring elements and that between the client and the servers is $2(\delta \cdot M + \mu) \cdot (\lambda \cdot \ell)$ bits with $(\delta \cdot M + \mu)$ ring elements.

In the semi-honest ReLU protocol, each sign check operation is followed with a multiplication. Similar to the optimizations we mentioned in Section A.1, we can pack each comparison and its corresponding multiplication together to verify. This also applies to the division protocol.

In logistic regression, after each iteration there will be $B$ sign checks that need to be verified. In our experiment for $n = 100,352$, we pick $\delta = \mu = 3, M = 8192$ according to the total number of sign checks, which means we run the protocol $\Pi^M_{\text{VerifySignCheck}}$ every 64 iterations. In an $m$-layer

**Protocol $\Pi^{M}_{\text{VerifySignCheck}}$:**

0. The two servers $\mathsf{S}_0$ and $\mathsf{S}_1$ and the client $\mathsf{C}$ agree on parameters $\delta, \mu$. Let $N := \delta \cdot M + \mu$.
1. For each $j \in [M]$, $t \in [\delta]$, let $k := \delta \cdot (j-1) + t$ and let the two servers do the following:
   (a) $\mathsf{S}_0$ samples a random $d \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ and sends $d$ to $\mathsf{S}_1$.
   (b) $\mathsf{S}_0$ sets $[\![a^k]\!]_0 = [\![x^j]\!]_0 + d$, $\widetilde{b}^{k,\mathsf{S}} = b^{j,\mathsf{S}}$, and $[\![\widetilde{b}^{k,\mathsf{C}}]\!]_0 = [\![b^{j,\mathsf{C}}]\!]_0$;
   $\mathsf{S}_1$ sets $[\![a^k]\!]_1 = [\![x^j]\!]_1 - d$, $\widetilde{b}^{k,\mathsf{S}} = b^{j,\mathsf{S}}$, and $[\![\widetilde{b}^{k,\mathsf{C}}]\!]_1 = [\![b^{j,\mathsf{C}}]\!]_1$.
2. For each $k \in \{\delta \cdot M + 1, \dots, \delta \cdot M + \mu\}$, the two servers do the following:
   (a) $\mathsf{S}_0$ samples a random $d \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ and sends $d$ to $\mathsf{S}_1$.
   (b) $\mathsf{S}_0$ sets $[\![a^k]\!]_0 = d$, $\widetilde{b}^{k,\mathsf{S}} = 0$, and $[\![\widetilde{b}^{k,\mathsf{C}}]\!]_0 = 0$.
   (c) $\mathsf{S}_1$ samples a random $r \in \mathbb{Z}_{2^\ell}$ such that $|r| < 2^{\ell_w + \ell_f}$ and sets $[\![a^k]\!]_1 = r - d$, $\widetilde{b}^{k,\mathsf{S}} = 0$, and $[\![\widetilde{b}^{k,\mathsf{C}}]\!]_1 = (r > 0)$.
3. Server $\mathsf{S}_0$ randomly samples a permutation $\pi : [N] \to [N]$ and sends it to $\mathsf{S}_1$.
4. Each server $\mathsf{S}_i$ ($i \in \{0,1\}$) uses $\pi$ to shuffle its tuples $\left\{\left([\![a^k]\!]_i, \widetilde{b}^{k,\mathsf{S}}, [\![\widetilde{b}^{k,\mathsf{C}}]\!]_i\right)\right\}_{k \in [N]}$ and obtain $\left\{\left([\![\hat{a}^k]\!]_i, \hat{b}^{k,\mathsf{S}}, [\![\hat{b}^{k,\mathsf{C}}]\!]_i\right)\right\}_{k \in [N]}$. In particular, $\left([\![\hat{a}^k]\!]_i, \hat{b}^{k,\mathsf{S}}, [\![\hat{b}^{k,\mathsf{C}}]\!]_i\right) := \left([\![a^{\pi(k)}]\!]_i, \widetilde{b}^{\pi(k),\mathsf{S}}, [\![\widetilde{b}^{\pi(k),\mathsf{C}}]\!]_i\right)$.
5. For each $k' \in \{\delta \cdot M + 1, \dots, \delta \cdot M + \mu\}$, let $k := \pi(k')$.
   (a) $\mathsf{S}_0, \mathsf{S}_1$ and $\mathsf{C}$ run $\Pi_{\text{SignCheck}}$ to compute
   $$((c^{k,\mathsf{S}}, [\![c^{k,\mathsf{C}}]\!]_0), (c^{k,\mathsf{S}}, [\![c^{k,\mathsf{C}}]\!]_1), c^{k,\mathsf{C}}) \leftarrow \Pi_{\text{SignCheck}}([\![\hat{a}^k]\!]_0, [\![\hat{a}^k]\!]_1, \bot).$$
   (b) The two servers $\mathsf{S}_0, \mathsf{S}_1$ run $\Pi_{\text{TwoSignCheck}}$ on $\left\{\left(\hat{b}^{k,\mathsf{S}}, c^{k,\mathsf{S}}, [\![\hat{b}^{k,\mathsf{C}}]\!]_i, [\![c^{k,\mathsf{C}}]\!]_i\right)\right\}_{i \in \{0,1\}}$, and abort the protocol if the output is not 0.
6. For each $j \in [M]$:
   (a) For each $t \in [\delta]$, let $k := \pi(\mu + \delta \cdot (j-1) + t)$, and let the two servers run $\Pi_{\text{SignCheck}}$ to compute
   $$((c^{k,\mathsf{S}}, [\![c^{k,\mathsf{C}}]\!]_0), (c^{k,\mathsf{S}}, [\![c^{k,\mathsf{C}}]\!]_1), c^{k,\mathsf{C}}) \leftarrow \Pi_{\text{SignCheck}}([\![\hat{a}^k]\!]_0, [\![\hat{a}^k]\!]_1, \bot),$$
   and then run $\Pi_{\text{TwoSignCheck}}$ on $\left\{\left(\hat{b}^{k,\mathsf{S}}, c^{k,\mathsf{S}}, [\![\hat{b}^{k,\mathsf{C}}]\!]_i, [\![c^{k,\mathsf{C}}]\!]_i\right)\right\}_{i \in \{0,1\}}$ to learn $e^j_t$.
   (b) If $e^j_t$ is not the same for all $t \in [\delta]$, then abort the protocol. If $e^j_t$ is the same for all $t \in [\delta]$ but is not 0, then output $(\mathsf{error}, j)$ along with $e^j_1$ and halt the protocol.
7. Both servers output $\mathsf{correct}$.

Figure 19: Protocol $\Pi^{M}_{\text{VerifySignCheck}}$ for verifying $M$ sign check results with the assistance of a client.

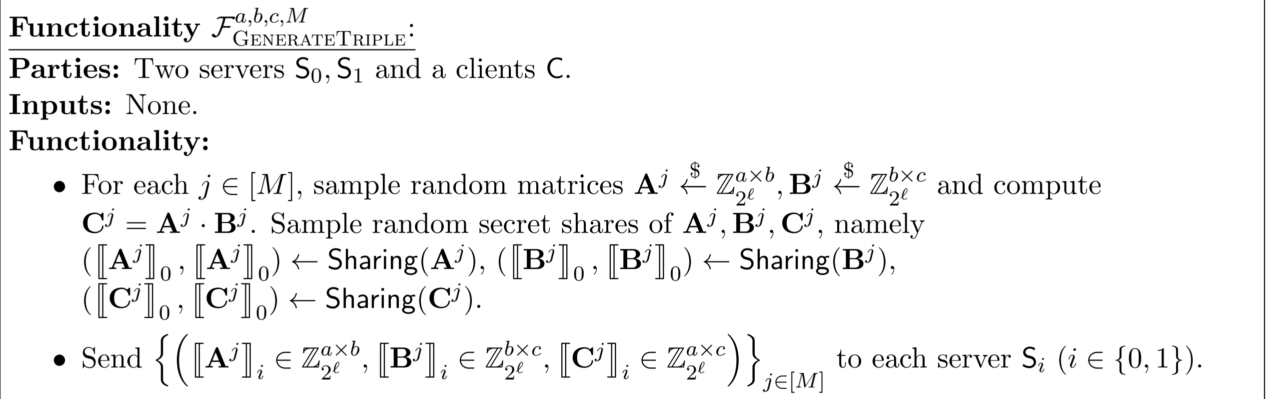
neural network, the number of sign checks is $B \cdot \sum d_i + B \cdot (\ell_f + 1) \cdot d_m$ ($i \in \{1, \ldots, m\}$) in each iteration. In our experiment for $n = 100,352$, we pick $\delta = \mu = 2, M = 1,048,576$.

### A.3 Client Generated Multiplication Triples

In neural networks, after the first layer, we use the same approach as the client-aided variant of [29], where the clients generate multiplication triples. In this section, we present a protocol to generate $M$ correct multiplication triples with the assistance of a potentially malicious client. The ideal functionality is described in Figure 20.

The protocol $\Pi^{a,b,c,M}_{\text{GenerateTriple}}$ is similar to protocol $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$ (presented in Section A.1). The main difference is that in $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$ the client generates $N$ new multiplication triples to verify whether the existing $M$ triples are correct, while in the new protocol $\Pi^{a,b,c,M}_{\text{GenerateTriple}}$ the client generates $N$ multiplication triples, uses them to verify each other, and finally selects $M$ correct triples from them. The protocol is described in Figure 21.

Note that an alternative approach is to first perform the multiplication using the triples generated by the client, and then perform the verification for the multiplication results using the protocol $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$. The difference between the two protocols that we discussed above makes it possible to achieve better parameters in $\Pi^{a,b,c,M}_{\text{GenerateTriple}}$.


**Functionality $\mathcal{F}^{a,b,c,M}_{\text{GenerateTriple}}$:**
**Parties:** Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a clients $\mathsf{C}$.
**Inputs:** None.
**Functionality:**

- For each $j \in [M]$, sample random matrices $\mathbf{A}^j \xleftarrow{\$} \mathbb{Z}_{2^\ell}^{a\times b}, \mathbf{B}^j \xleftarrow{\$} \mathbb{Z}_{2^\ell}^{b\times c}$ and compute $\mathbf{C}^j = \mathbf{A}^j \cdot \mathbf{B}^j$. Sample random secret shares of $\mathbf{A}^j, \mathbf{B}^j, \mathbf{C}^j$, namely $(\llbracket \mathbf{A}^j \rrbracket_0, \llbracket \mathbf{A}^j \rrbracket_0) \leftarrow \mathsf{Sharing}(\mathbf{A}^j)$, $(\llbracket \mathbf{B}^j \rrbracket_0, \llbracket \mathbf{B}^j \rrbracket_0) \leftarrow \mathsf{Sharing}(\mathbf{B}^j)$, $(\llbracket \mathbf{C}^j \rrbracket_0, \llbracket \mathbf{C}^j \rrbracket_0) \leftarrow \mathsf{Sharing}(\mathbf{C}^j)$.
- Send $\left\{ \left( \llbracket \mathbf{A}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times b}, \llbracket \mathbf{B}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{b\times c}, \llbracket \mathbf{C}^j \rrbracket_i \in \mathbb{Z}_{2^\ell}^{a\times c} \right) \right\}_{j\in[M]}$ to each server $\mathsf{S}_i$ ($i \in \{0, 1\}$).

Figure 20: Ideal functionality $\mathcal{F}^{a,b,c,M}_{\text{GenerateTriple}}$ for generating of $M$ multiplication triples.

**Theorem A.5.** *For the parameters we choose below, the protocol $\Pi^{a,b,c,M}_{\text{GenerateTriple}}$ (Figure 21) securely computes the ideal functionality $\mathcal{F}^{a,b,c,M}_{\text{GenerateTriple}}$ (Figure 20) against an adversary that corrupts either one of the two servers $\mathsf{S}_0, \mathsf{S}_1$ in a semi-honest way, or the client $\mathsf{C}$ maliciously.*

**Proof Sketch and Parameters.** Our protocol is very similar to PROTOCOL 3.2 (Generating Multiplication triples) in [12], except that they assume three parties with one malicious party while we assume two semi-honest servers along with a malicious client. The security proof and the analysis for the parameters are almost the same as [12], and we refer the reader to that paper for more details. Here we only present the result.

Let $t$ be the number of buckets $\mathcal{A}$ has chosen to corrupt. Then for every $0 < t \leq M$, it holds that

$$\Pr\left[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1\right] = \binom{M}{t} \binom{\delta \cdot M + \mu}{t\delta}^{-1}.$$

**Protocol** $\Pi^{a,b,c,M}_{\textsc{GenerateTriple}}$:

0. The two servers $\mathsf{S}_0$ and $\mathsf{S}_1$ and the client $\mathsf{C}$ agree on parameters $\delta, \mu$. Let $N := \delta \cdot M + \mu$.
1. The client $\mathsf{C}$ does the following:
   (a) For each $k \in [N]$, sample random matrices $\mathbf{A}^k \xleftarrow{\$} \mathbb{Z}^{a\times b}_{2^\ell}, \mathbf{B}^k \xleftarrow{\$} \mathbb{Z}^{b\times c}_{2^\ell}$ and compute $\mathbf{C}^k = \mathbf{A}^k \cdot \mathbf{B}^k$.
   (b) For each $k \in [N]$, generate additive secret sharings $(\llbracket \mathbf{A}^k \rrbracket_0, \llbracket \mathbf{A}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{A}^k), (\llbracket \mathbf{B}^k \rrbracket_0, \llbracket \mathbf{B}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{B}^k), (\llbracket \mathbf{C}^k \rrbracket_0, \llbracket \mathbf{C}^k \rrbracket_1) \leftarrow \mathsf{Sharing}(\mathbf{C}^k)$.
   (c) Send $\{(\llbracket \mathbf{A}^k \rrbracket_i, \llbracket \mathbf{B}^k \rrbracket_i, \llbracket \mathbf{C}^k \rrbracket_i)\}_{k\in[N]}$ to each server $\mathsf{S}_i$ $(i \in \{0, 1\})$.
2. Server $\mathsf{S}_0$ randomly samples a permutation $\pi : [N] \to [N]$ and sends it to $\mathsf{S}_1$.
3. Each server $\mathsf{S}_i$ $(i \in \{0, 1\})$ uses $\pi$ to shuffle its triples and obtain $\left\{\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)\right\}_{k\in[N]}$. In particular, $\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right) := (\llbracket \mathbf{A}^{\pi(k)} \rrbracket_i, \llbracket \mathbf{B}^{\pi(k)} \rrbracket_i, \llbracket \mathbf{C}^{\pi(k)} \rrbracket_i)$.
4. The two servers run $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ on each of the first $\mu$ shuffled triples, namely $\left\{\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)\right\}_{k=1}^{\mu}$. The servers abort the protocol if any $\Pi^{a,b,c}_{\textsc{OpenTriple}}$ instance outputs 0.
5. For each $j \in [M]$, let $l := \mu + \delta \cdot (j - 1) + 1$.
   (a) For each $t \in [\delta - 1]$, let $k := \mu + \delta \cdot (j - 1) + t + 1$, and let the two servers run $\Pi^{a,b,c}_{\textsc{TwoTriples}}$ on $\left(\left\llbracket \hat{\mathbf{A}}^l \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^l \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^l \right\rrbracket_i\right)$ and $\left(\left\llbracket \hat{\mathbf{A}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^k \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^k \right\rrbracket_i\right)$ to learn $\mathbf{D}^j_t$.
   (b) If $\mathbf{D}^j_t = \mathbf{0}$ for all $t \in [\delta - 1]$, then each server $\mathsf{S}_i$ stores $\mathbf{O}^j_i := \left(\left\llbracket \hat{\mathbf{A}}^l \right\rrbracket_i, \left\llbracket \hat{\mathbf{B}}^l \right\rrbracket_i, \left\llbracket \hat{\mathbf{C}}^l \right\rrbracket_i\right)$, namely they store these shares in the $j$-th entry of $\mathbf{O}_i$. Otherwise abort the protocol.
6. Each server $\mathsf{S}_i$ outputs $\left\{\mathbf{O}^j_i\right\}_{j\in[M]}$.

Figure 21: Protocol $\Pi^{a,b,c,M}_{\textsc{GenerateTriple}}$ for generating $M$ multiplication triples.

| $\delta$ | $\mu$ | $M$ | $N = \delta \cdot M + \mu$ |
|---|---|---|---|
| 6 | 6 | 128($2^7$) | 774 |
| 5 | 5 | 512($2^9$) | 2,565 |
| 4 | 4 | 8,192($2^{13}$) | 32,772 |
| 3 | 3 | 524,288($2^{19}$) | 1,572,867 |

Table 11: Parameters for generating multiplication triples for $\sigma = 40$.

If $\mu \geq \delta$, then for every adversary $\mathcal{A}$, it holds that

$$\Pr[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1] \leq M \binom{\delta \cdot M + \mu}{\delta}^{-1}.$$

If $\mu = \delta$ and $\delta, M$ are chosen such that $\sigma \leq \log\left(\frac{(\delta \cdot M + \delta)!}{\delta \cdot M! \cdot (\delta \cdot M)!}\right)$, then for every adversary $\mathcal{A}$ it holds

that $\Pr[\mathsf{Game}(\mathcal{A}, M, \delta, \mu) = 1] \leq 2^{-\sigma}$. In Table 11, we present the minimum $M$ (power of 2) for $\sigma = 40$ and different $\delta$ and $\mu$ values.

**Communication and Optimizations.** Similar to the protocol $\Pi^{a,b,c,M}_{\text{VerifyTriple}}$, the communication of this protocol between the two servers is $2(M \cdot (\delta - 1) + \mu) \cdot (a \cdot b + b \cdot c) + 2(\mu + 1) \cdot (a \cdot c)$ ring elements and that between the client and the servers is $(\delta \cdot M + \mu) \cdot (a \cdot c)$ ring elements.

In neural network training, we use the same optimizations as in Section A.1 to combine $2B$ multiplication triples into one triple. Considering the total number of the triples need to generate, we select $\delta = \mu = 5$ and $M = 512$, which means we call the protocol $\Pi^{a,b,c,M}_{\text{GenerateTriple}}$ every 512 iterations.

# B Deferred Security Proofs

## B.1 Proof of Theorem 4.1

**Corrupted $\mathsf{C}$:** In the case of a semi-honest client, the construction of its simulator is trivial as its view contains only messages it sends. We only need to prove the correctness of the protocol, namely by executing the protocol the output is consistent with the ideal functionality. Upon receiving $(\llbracket \mathbf{r} \rrbracket_i, \llbracket u \rrbracket_i)$ from the client, the servers first compute $\llbracket \mathbf{s} \rrbracket_i = \llbracket \mathbf{x} \rrbracket_i - \llbracket \mathbf{r} \rrbracket_i$, and then recover $\mathbf{s} = \llbracket \mathbf{s} \rrbracket_0 + \llbracket \mathbf{s} \rrbracket_1 = \llbracket \mathbf{x} \rrbracket_0 + \llbracket \mathbf{x} \rrbracket_1 - (\llbracket \mathbf{r} \rrbracket_0 + \llbracket \mathbf{r} \rrbracket_1) = \mathbf{x} - \mathbf{r}$. Hence $\llbracket v \rrbracket_i = \langle \mathbf{s}, \llbracket \mathbf{y} \rrbracket_i \rangle + \llbracket u \rrbracket_i = \langle \mathbf{x} - \mathbf{r}, \llbracket \mathbf{y} \rrbracket_i \rangle + \llbracket u \rrbracket_i$. Note that $u = \langle \mathbf{r}, \mathbf{y} \rangle$, so we have $\llbracket v \rrbracket_0 + \llbracket v \rrbracket_1 = (\langle \mathbf{x} - \mathbf{r}, \llbracket \mathbf{y} \rrbracket_0 \rangle + \llbracket u \rrbracket_0) + (\langle \mathbf{x} - \mathbf{r}, \llbracket \mathbf{y} \rrbracket_1 \rangle + \llbracket u \rrbracket_1) = \langle \mathbf{x} - \mathbf{r}, \mathbf{y} \rangle + \langle \mathbf{r}, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{y} \rangle$. Since $\llbracket u \rrbracket_0$ and $\llbracket u \rrbracket_1$ are additive secret shares of $u$, the outputs $\llbracket v \rrbracket_0$ and $\llbracket v \rrbracket_1$ are additive secret shares of $v$, which is the same distribution as the ideal functionality.

**Corrupted $\mathsf{S}_0$:** In the case of a semi-honest server, assume without loss of generality that the adversary corrupts $\mathsf{S}_0$ as the two servers are symmetric in the protocol. We construct a simulator $\mathcal{S}$ to simulate its view. We construct $\mathcal{S}$ given inputs $(\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0)$ and output $\llbracket v \rrbracket_0$ as follows:

1. Sample uniform random vectors $\llbracket \mathbf{r} \rrbracket_0 \xleftarrow{\$} \mathbb{Z}^d_{2^\ell}$ in Step 1d and $\llbracket \mathbf{s} \rrbracket_1 \xleftarrow{\$} \mathbb{Z}^d_{2^\ell}$ in Step 2a.
2. Set $\llbracket u \rrbracket_0 := \llbracket v \rrbracket_0 - \langle \llbracket \mathbf{x} \rrbracket_0 - \llbracket \mathbf{r} \rrbracket_0 + \llbracket \mathbf{s} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_0 \rangle$ in Step 1d.
3. Follow the protocol description of $\mathsf{S}_0$ and output its view.

Next we prove that for any inputs $(\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), (\llbracket \mathbf{x} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_1), \llbracket \mathbf{y} \rrbracket$,

$$\begin{aligned}&\left(\mathsf{View}^{\Pi}_{\mathsf{S}_0}((\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), (\llbracket \mathbf{x} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_1), \llbracket \mathbf{y} \rrbracket), \mathsf{Out}^{\Pi}_{\mathsf{S}_1}((\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), (\llbracket \mathbf{x} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_1), \llbracket \mathbf{y} \rrbracket)\right) \\ &\overset{s}{\equiv} \left(\mathcal{S}\left(1^\lambda, (\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), f_{\mathsf{S}_0}((\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), (\llbracket \mathbf{x} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_1), \llbracket \mathbf{y} \rrbracket)\right), f_{\mathsf{S}_1}((\llbracket \mathbf{x} \rrbracket_0, \llbracket \mathbf{y} \rrbracket_0), (\llbracket \mathbf{x} \rrbracket_1, \llbracket \mathbf{y} \rrbracket_1), \llbracket \mathbf{y} \rrbracket)\right)\end{aligned}$$

via the following hybrid argument, where $\overset{s}{\equiv}$ denotes that the two distributions are statistically identical.

$\mathsf{Hyb}_0$ $\mathsf{S}_0$'s view along with $\mathsf{S}_1$'s output in the real-world protocol execution.

$\mathsf{Hyb}_1$ Same as $\mathsf{Hyb}_0$ except that we randomly sample $\llbracket \mathbf{s} \rrbracket_1 \xleftarrow{\$} \mathbb{Z}^d_{2^\ell}$ and set $\llbracket \mathbf{r} \rrbracket_1 := \llbracket \mathbf{x} \rrbracket_1 - \llbracket \mathbf{s} \rrbracket_1$. This hybrid is statistically identical to $\mathsf{Hyb}_0$.

$\mathsf{Hyb}_2$ Same as $\mathsf{Hyb}_1$ except that we replace $\mathsf{S}_1$'s output with $\llbracket v \rrbracket_1 = \langle \mathbf{x}, \mathbf{y} \rangle - \llbracket v \rrbracket_0$. This hybrid is statistically identical to $\mathsf{Hyb}_1$ because of the perfect correctness of the protocol that we show above.

$\mathsf{Hyb}_3$ Same as $\mathsf{Hyb}_2$ except that we randomly sample $[\![v]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ and set $[\![u]\!]_0 := [\![v]\!]_0 - \langle \mathbf{s}, [\![\mathbf{y}]\!]_0 \rangle$ in Step 1d. This hybrid is statistically identical to $\mathsf{Hyb}_2$ because $[\![u]\!]_0$ is uniform randomly sampled in $\mathsf{Sharing}(u)$.

$\mathsf{Hyb}_4$ Same as $\mathsf{Hyb}_3$ except that we sample uniform random $[\![\mathbf{r}]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}^d$ in Step 1d. This hybrid is statistically identical to $\mathsf{Hyb}_3$, and is exactly $\mathcal{S}$'s output along with $\mathsf{S}_1$'s output in the ideal world.

## B.2 Proof of Theorem 4.2

**Corrupted $\mathsf{C}$:** For security against a semi-honest client, we construct $\mathcal{S}_\mathsf{C}$ as follows. Given the client's output $b^\mathsf{C}$, if $b^\mathsf{C} = 0$, then $\mathcal{S}_\mathsf{C}$ samples two sets $\widetilde{\mathcal{T}_0}, \widetilde{\mathcal{T}_1}$, each of size $\ell$, where all the elements are sampled from a uniform distribution over $\{0,1\}^\lambda$; if $b^\mathsf{C} = 1$, then $\mathcal{S}_\mathsf{C}$ samples the two sets $\widetilde{\mathcal{T}_0}, \widetilde{\mathcal{T}_1}$ in the same way except that they share a common element. $\mathcal{S}_\mathsf{C}$ follows the protocol description of $\mathsf{C}$, sets $(\widetilde{\mathcal{T}_0}, \widetilde{\mathcal{T}_1})$ as the two sets received from the servers in Step 2d, and output the view of $\mathsf{C}$.

Next we prove that for any $[\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1 \in \mathbb{Z}_{2^\ell}$ that have different signs,

$$\left(\mathsf{View}_\mathsf{C}^\Pi\left([\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1, \bot\right), \mathsf{Out}_{\mathsf{S}_0,\mathsf{S}_1}^\Pi\left([\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1, \bot\right)\right) \overset{c}{\approx} \left(\mathcal{S}_\mathsf{C}\left(1^\lambda, f_\mathsf{C}\left([\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1, \bot\right)\right), f_{\mathsf{S}_0,\mathsf{S}_1}\left([\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1, \bot\right)\right)$$

via the following hybrid argument, where $\overset{c}{\approx}$ denotes that the two distributions are computationally indistinguishable.

$\mathsf{Hyb}_0$ $\mathsf{C}$'s view along with the two servers' outputs in the real-world protocol execution.

$\mathsf{Hyb}_1$ Same as $\mathsf{Hyb}_0$ except that we replace the PRF $F_k$ by a truly random function. This hybrid is computationally indistinguishable to $\mathsf{Hyb}_0$ because otherwise we can break the security of PRF. Specifically, if there exists a PPT $\mathcal{D}$ that can distinguish $\mathsf{Hyb}_1$ from $\mathsf{Hyb}_0$, then we can construct a PPT $\mathcal{B}$ that breaks the PRF security. The adversary $\mathcal{B}$ runs the protocol as in $\mathsf{Hyb}_0$ but when it needs to compute $F_k(x)$ for some $x$, it queries the PRF challenger $\mathcal{C}$ on $x$. Finally $\mathcal{B}$ runs $\mathcal{D}$ on the client $\mathsf{C}$'s view along with the two servers' outputs. If $\mathcal{C}$ returns outputs of a $F_k$, then $\mathcal{D}$ receives $\mathsf{Hyb}_0$; otherwise $\mathcal{D}$ receives $\mathsf{Hyb}_1$. If $\mathcal{D}$ can distinguish $\mathsf{Hyb}_0$ and $\mathsf{Hyb}_1$, then $\mathcal{B}$ can distinguish a PRF and a truly random function.

$\mathsf{Hyb}_2$ Note that our construction of augmented sets guarantees that there is at most common element in $(\mathbf{A}_0, \mathbf{A}_1)$. $\mathsf{Hyb}_2$ is the same as $\mathsf{Hyb}_1$ except we replace $\widetilde{\mathcal{T}_0}, \widetilde{\mathcal{T}_1}$ by randomly sampled sets. In particular, if $|\mathbf{A}_0 \cap \mathbf{A}_1| = 0$, then we sample all the elements in $\widetilde{\mathcal{T}_0}, \widetilde{\mathcal{T}_1}$ from a uniform distribution over $\{0,1\}^\lambda$; if $|\mathbf{A}_0 \cap \mathbf{A}_1| = 1$, then we sample all the elements in the same way except that we reuse a value in $\widetilde{\mathcal{T}_0}$ and $\widetilde{\mathcal{T}_1}$. This hybrid is statistically identical to $\mathsf{Hyb}_1$ because of the way a random function works.

$\mathsf{Hyb}_3$ Same as $\mathsf{Hyb}_2$ except that $|\mathbf{A}_0 \cap \mathbf{A}_1|$ is computed as $(x > 0) \oplus b^\mathsf{S}$. This hybrid is statistically identical to $\mathsf{Hyb}_2$, which follows from the construction of augmented sets and the correctness of our protocol.

For two non-negative values $a, b \in \mathbb{Z}_{2^\ell}$, if we let $\mathbf{A}_a^1$ be the augmented 1-encoding of $a$ and let $\mathbf{A}_b^0$ be the augmented 0-encoding of $b$, then $a > b$ iff $|\mathbf{A}_a^1 \cap \mathbf{A}_b^0| = 1$ and $a \leq b$ iff $|\mathbf{A}_a^1 \cap \mathbf{A}_b^0| = 0$.

Without loss of generality we assume $[\![x]\!]_0 \geq 0$ and $[\![x]\!]_1 < 0$. $\mathsf{S}_0$ generates $\mathbf{A}_0$ as an augmented $(1 \oplus b^\mathsf{S})$-encoding of $[\![x]\!]_0$ and $\mathsf{S}_1$ generates $\mathbf{A}_1$ as an augmented $b^\mathsf{S}$-encoding of $-[\![x]\!]_1$.

If $b^{\mathsf{S}} = 0$, then $[\![x]\!]_0 > -[\![x]\!]_1$ iff $|\mathbf{A}_0 \cap \mathbf{A}_1| = 1$ and $[\![x]\!]_0 \leq -[\![x]\!]_1$ iff $|\mathbf{A}_0 \cap \mathbf{A}_1| = 0$, hence $|\mathbf{A}_0 \cap \mathbf{A}_1| = (x > 0)$. If $b^{\mathsf{S}} = 1$, then $-[\![x]\!]_1 > [\![x]\!]_0$ iff $|\mathbf{A}_0 \cap \mathbf{A}_1| = 1$ and $-[\![x]\!]_1 \leq [\![x]\!]_0$ iff $|\mathbf{A}_0 \cap \mathbf{A}_1| = 0$, hence $|\mathbf{A}_0 \cap \mathbf{A}_1| = (x > 0) \oplus 1$. In both cases, $|\mathbf{A}_0 \cap \mathbf{A}_1| = (x > 0) \oplus b^{\mathsf{S}}$.

$\mathsf{Hyb}_4$ Same as $\mathsf{Hyb}_3$ except that we replace $|\mathbf{A}_0 \cap \mathbf{A}_1|$ by the client's output $b^{\mathsf{C}}$ in the ideal world and replace the server's outputs with their outputs in the ideal world. This hybrid is statistically identical to $\mathsf{Hyb}_3$, and is exactly $\mathcal{S}_{\mathsf{C}}$'s output along with the two servers' outputs in the ideal world.

**Corrupted $\mathsf{S}_0$:** For a semi-honest adversary that corrupts one of the two servers, we consider the case where the adversary corrupts $\mathsf{S}_0$, and the proof for a corrupted $\mathsf{S}_1$ is almost identical. Simulation for $\mathsf{S}_0$ is simple as we only need to randomly sample $[\![b^{\mathsf{C}}]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ in Step 4, which is a random share of $b^{\mathsf{C}}$. The rest of the proof follows from the correctness of the protocol, which is shown above.

## B.3 Proof of Theorem 4.3

**Corrupted $\mathsf{C}$:** For a semi-honest adversary that corrupts the client $\mathsf{C}$, the construction of its simulator follows the protocol as an honest client, and only needs to sample a random $b^{\mathsf{C}} \xleftarrow{\$} \{0,1\}$ as the output from $\mathcal{F}_{\text{SignCheck}}$ in Step 1. The simulated view of $\mathsf{C}$ is statistically identical to its view in $\Pi_{\text{ReLU}}$.

To prove security in this case, it only remains to show correctness of the protocol, namely by executing the protocol the outputs of the servers are consistent with the ideal functionality. Let $b := ([\![x]\!]_0 + [\![x]\!]_1 > 0)$, namely a predicate of whether $x$ is positive or not. Note that $\text{ReLU}(x) = x \cdot b$. From $\mathcal{F}_{\text{SignCheck}}$, it holds that $b^{\mathsf{S}} \oplus b^{\mathsf{C}} = b$.

If $b^{\mathsf{S}} = 0$, then $\text{ReLU}(x) = x \cdot b^{\mathsf{C}}$. The servers learn a random secret sharing of $x \cdot b^{\mathsf{C}}$, in particular $([\![\alpha]\!]_0, [\![\alpha]\!]_1)$, from $\mathcal{F}^1_{\text{InnerProd}}$. Then they compute $[\![y]\!]_i = [\![\alpha]\!]_i$, which results in a secret sharing of $x \cdot b^{\mathsf{C}}$.

If $b^{\mathsf{S}} = 1$, then $\text{ReLU}(x) = x \cdot (1 - b^{\mathsf{C}}) = x - x \cdot b^{\mathsf{C}}$. Again, the servers learn a random secret sharing of $x \cdot b^{\mathsf{C}}$, in particular $([\![\alpha]\!]_0, [\![\alpha]\!]_1)$, from $\mathcal{F}^1_{\text{InnerProd}}$. Then they compute $[\![y]\!]_i = [\![x]\!]_i - [\![\alpha]\!]_i$, which results in a secret sharing of $x - x \cdot b^{\mathsf{C}}$.

In both cases, the two servers output random secret shares of $\text{ReLU}(x)$, which is statistically identical to the ideal functionality.

**Corrupted $\mathsf{S}_0$:** For a semi-honest adversary that corrupts one of the two servers, assume without loss of generality that the adversary corrupts $\mathsf{S}_0$ as the two servers are symmetric in the protocol. We construct a simulator $\mathcal{S}$ to simulate its view. Given the input $[\![x]\!]_0$ and output $[\![y]\!]_0$ from the ideal functionality $\mathcal{F}_{\text{ReLU}}$, $\mathcal{S}$ is constructed as follows:

1. Sample uniform random $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and $[\![b^{\mathsf{C}}]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}_{\text{SignCheck}}$.
2. If $b^{\mathsf{S}} = 0$, then let $[\![\alpha]\!]_0 := [\![y]\!]_0$; otherwise compute $[\![\alpha]\!]_0 := [\![x]\!]_0 - [\![y]\!]_0$. Set $[\![\alpha]\!]_0$ as the output from $\mathcal{F}^1_{\text{InnerProd}}$.
3. Follow the protocol description of $\mathsf{S}_0$ and output its view.

Next we prove that for any $[\![x]\!]_0, [\![x]\!]_1 \in \mathbb{Z}^d_{2^\ell}$,

$$\left(\mathsf{View}^{\Pi}_{\mathsf{S}_0}([\![x]\!]_0, [\![x]\!]_1, \perp), \mathsf{Out}^{\Pi}_{\mathsf{S}_1}([\![x]\!]_0, [\![x]\!]_1, \perp)\right) \overset{s}{\equiv} \left(\mathcal{S}\left(1^\lambda, [\![x]\!]_0, f_{\mathsf{S}_0}([\![x]\!]_0, [\![x]\!]_1, \perp)\right), f_{\mathsf{S}_1}([\![x]\!]_0, [\![x]\!]_1, \perp)\right)$$

via the following hybrid argument.

$\mathsf{Hyb}_0$ $\mathsf{S}_0$'s view along with $\mathsf{S}_1$'s output in the real-world protocol execution.

$\mathsf{Hyb}_1$ Same as $\mathsf{Hyb}_0$ except that $\mathsf{S}_1$'s output is replaced with $[\![y]\!]_1 = \max\{x, 0\} - [\![y]\!]_0$. This hybrid is statistically identical to $\mathsf{Hyb}_1$ because of the perfect correctness of the protocol that we show above.

$\mathsf{Hyb}_2$ Same as $\mathsf{Hyb}_1$ except that we first sample $[\![y]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ and then compute $[\![\alpha]\!]_0$ in Step 2 as follows: if $b^{\mathsf{S}} = 0$, then let $[\![\alpha]\!]_0 := [\![y]\!]_0$; otherwise compute $[\![\alpha]\!]_0 := [\![x]\!]_0 - [\![y]\!]_0$. This hybrid is statistically identical to $\mathsf{Hyb}_1$.

$\mathsf{Hyb}_3$ Same as $\mathsf{Hyb}_2$ except that in Step 1 we sample uniform random $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and $[\![b^{\mathsf{C}}]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}_{\text{SignCheck}}$. This hybrid is statistically identical to $\mathsf{Hyb}_2$, which follows from the functionality of $\mathcal{F}_{\text{SignCheck}}$. This hybrid is exactly $\mathcal{S}$'s output along with $\mathsf{S}_1$'s output in the ideal world.

### B.4 Proof of Theorem 4.4

**Corrupted $\mathsf{C}$:** For a semi-honest adversary that corrupts the client $\mathsf{C}$, the construction of its simulator follows the protocol as an honest client, and only needs to sample a random $b_j^{\mathsf{C}} \xleftarrow{\$} \{0,1\}$ as the output from $\mathcal{F}_{\text{SignCheck}}$ in Step 2b for all $j$. The simulated view of $\mathsf{C}$ is statistically identical to its view in $\Pi_{\text{Div}}$. To prove security in this case, it only remains to show correctness of the protocol, namely by executing the protocol the outputs of the servers are consistent with the ideal functionality.

The division is done bit by bit for $(\ell_f + 1)$ times. We show that for each $j$ from $\ell_f$ downto 0, $([\![u_{j+1}]\!]_0, [\![u_{j+1}]\!]_1)$ is a secret sharing of the dividend in that step, and $([\![k_j]\!]_0, [\![k_j]\!]_1)$ is a secret sharing of the quotient bit in that step.

We can prove this by induction. First, $[\![u_{\ell_f+1}]\!]_i$ is initialized as $[\![x]\!]_i$, which is the dividend to be used for $j = \ell_f$. For each $j$ from $\ell_f$ downto 0, let $b_j := (u_{j+1} - y \geq 0)$, namely a predicate of whether the dividend is greater than or equal to $y$. In fact, the quotient bit in this step should be exactly $b_j$ and the remainder should be $u_{j+1} - y \cdot b_j$.

Note that $([\![v_j]\!]_0, [\![v_j]\!]_1)$ is a secret sharing of $u_{j+1} - y + 1$. From $\mathcal{F}_{\text{SignCheck}}$, it holds that $b_j^{\mathsf{S}} \oplus b_j^{\mathsf{C}} = b_j$. Next, from $\mathcal{F}^1_{\text{InnerProd}}$ the servers learn $([\![v_j]\!]_0, [\![v_j]\!]_1)$, which are random secret shares of $v_j = y \cdot b_j^{\mathsf{C}}$. Similarly as in the proof of Theorem 4.3, we can analyze the two cases of $b_j^{\mathsf{S}} = 0$ and $b_j^{\mathsf{S}} = 1$. We omit the details here but in both cases, the two servers learn $([\![k_j]\!]_0, [\![k_j]\!]_1)$, which are secret shares of $k_j = b_j^{\mathsf{S}} + b_j^{\mathsf{C}} \cdot (1 - 2b_j^{\mathsf{S}}) = b_j$, exactly the quotient in this step. They also learn $\left([\![v_j^*]\!]_0, [\![v_j^*]\!]_1\right)$, which are secret shares of $v_j^* = y \cdot b_j$, and $([\![u_j]\!]_0, [\![u_j]\!]_1)$, which are secret shares of $2 \cdot (u_{j+1} - y \cdot b_j)$, exactly the dividend to be used in the next step.

Finally, the quotient $q$ is computed as $\sum_{j=0}^{\ell_f} 2^j \cdot k_j$, in a secret shared manner.

**Corrupted $\mathsf{S}_0$:** For a semi-honest adversary that corrupts one of the two servers, we consider the case where the adversary corrupts $\mathsf{S}_0$, and the proof for a corrupted $\mathsf{S}_1$ is almost identical. We construct a simulator $\mathcal{S}$ to simulate its view. Given the input $([\![x]\!]_0, [\![y]\!]_0)$ and output $[\![q]\!]_0$ from the ideal functionality $\mathcal{F}_{\text{Div}}$, $\mathcal{S}$ is constructed as follows:

1. For $j$ from $\ell_f$ downto 1: sample uniform random $b_j^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and $[\![b_j^{\mathsf{C}}]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the

output from $\mathcal{F}_{\textsc{SignCheck}}$ in Step 2b; sample uniform random $[\![v_j]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}^1_{\textsc{InnerProd}}$ in Step 2c; compute $[\![k_j]\!]_0 := \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0 \cdot (1 - 2b^{\mathsf{S}}_j)$.

2. For $j = 0$: sample uniform random $b^{\mathsf{S}}_0 \xleftarrow{\$} \{0,1\}$, compute $\left[\!\!\left[b^{\mathsf{C}}_0\right]\!\!\right]_0 := ([\![q]\!]_0 - \sum_{j=1}^{\ell_f} 2^j \cdot [\![k_j]\!]_0)/(1 - 2b^{\mathsf{S}}_0)$, and set $\left(b^{\mathsf{S}}_0, \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0\right)$ as the output from $\mathcal{F}_{\textsc{SignCheck}}$ in Step 2b; sample uniform random $[\![v_0]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}^1_{\textsc{InnerProd}}$ in Step 2c.
3. Follow the protocol description of $\mathsf{S}_0$ and output its view.

Next we prove that for any $[\![x]\!]_0, [\![x]\!]_1 \in \mathbb{Z}^d_{2^\ell}$,

$$\left(\mathsf{View}^{\Pi}_{\mathsf{S}_0}([\![x]\!]_0, [\![x]\!]_1, \bot), \mathsf{Out}^{\Pi}_{\mathsf{S}_1}([\![x]\!]_0, [\![x]\!]_1, \bot)\right) \overset{s}{\equiv} \left(\mathcal{S}\left(1^\lambda, [\![x]\!]_0, f_{\mathsf{S}_0}([\![x]\!]_0, [\![x]\!]_1, \bot)\right), f_{\mathsf{S}_1}([\![x]\!]_0, [\![x]\!]_1, \bot)\right)$$

via the following hybrid argument.

$\mathsf{Hyb}_0$ $\mathsf{S}_0$'s view along with $\mathsf{S}_1$'s output in the real-world protocol execution.

$\mathsf{Hyb}_1$ Same as $\mathsf{Hyb}_0$ except that $\mathsf{S}_1$'s output is replaced with $[\![q]\!]_1 = \mathsf{Quotient}(x, y) - [\![q]\!]_0$. This hybrid is statistically identical to $\mathsf{Hyb}_1$ because of the perfect correctness of the protocol that we show above.

$\mathsf{Hyb}_2$ Same as $\mathsf{Hyb}_1$ except that we first sample $[\![q]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ and then compute $\left(b^{\mathsf{S}}_j, \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0\right)$ in Step 2b as follows. For $j$ from $\ell_f$ downto 1, sample uniform random $b^{\mathsf{S}}_j \xleftarrow{\$} \{0,1\}$ and $\left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}_{\textsc{SignCheck}}$, and compute $[\![k_j]\!]_0 := \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0 \cdot (1 - 2b^{\mathsf{S}}_j)$. For $j = 0$, sample uniform random $b^{\mathsf{S}}_0 \xleftarrow{\$} \{0,1\}$, compute $\left[\!\!\left[b^{\mathsf{C}}_0\right]\!\!\right]_0 := ([\![q]\!]_0 - \sum_{j=1}^{\ell_f} 2^j \cdot [\![k_j]\!]_0)/(1 - 2b^{\mathsf{S}}_0)$, and set $\left(b^{\mathsf{S}}_0, \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0\right)$ as the output from $\mathcal{F}_{\textsc{SignCheck}}$. The indistinguishability of $\mathsf{Hyb}_2$ and $\mathsf{Hyb}_1$ follows from the functionality of $\mathcal{F}_{\textsc{SignCheck}}$, namely $\left(b^{\mathsf{S}}_j, \left[\!\!\left[b^{\mathsf{C}}_j\right]\!\!\right]_0\right)$ are all random shares. Note that $[\![k_0]\!]_0$ randomly masks the secret share $[\![q]\!]_0$ in the protocol, hence this hybrid is statistically identical to $\mathsf{Hyb}_1$.

$\mathsf{Hyb}_3$ Same as $\mathsf{Hyb}_2$ except that in Step 2c we sample uniform random $[\![v_j]\!]_0 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ as the output from $\mathcal{F}^1_{\textsc{InnerProd}}$ for all $j$. This hybrid is statistically identical to $\mathsf{Hyb}_2$ because of the functionality of $\mathcal{F}^1_{\textsc{InnerProd}}$, and is exactly $\mathcal{S}$'s output along with $\mathsf{S}_1$'s output in the ideal world.