pMPL: A Robust Multi-Party Learning Framework with a Privileged Party

Lushan Song
Fudan University
19110240022@fudan.edu.cn

Jiaxuan Wang
Fudan University
20212010090@fudan.edu.cn

Zhexuan Wang
Fudan University
21210240331@m.fudan.edu.cn

Xinyu Tu
Fudan University
21210240326@m.fudan.edu.cn

Guopeng Lin
Fudan University
17302010022@fudan.edu.cn

Wenqiang Ruan
Fudan University
20110240031@fudan.edu.cn

Haoqi Wu
Fudan University
19212010008@fudan.edu.cn

Weili Han
Fudan University
wlhan@fudan.edu.cn

ABSTRACT

In order to perform machine learning among multiple parties while protecting the privacy of raw data, privacy-preserving machine learning based on secure multi-party computation (MPL for short) has been a hot spot in recent. The configuration of MPL usually follows the peer-to-peer architecture, where each party has the same chance to reveal the output result. However, typical business scenarios often follow a hierarchical architecture where a powerful, usually *privileged party*, leads the tasks of machine learning. Only the *privileged party* can reveal the final model even if other *assistant parties* collude with each other. It is even required to avoid the abort of machine learning to ensure the scheduled deadlines and/or save used computing resources when part of *assistant parties* drop out.

Motivated by the above scenarios, we propose pMPL, a robust MPL framework with a *privileged party*. pMPL supports three-party (a typical number of parties in MPL frameworks) training in the semi-honest setting. By setting alternate shares for the *privileged party*, pMPL is robust to tolerate one of the rest two parties dropping out during the training. With the above settings, we design a series of efficient protocols based on vector space secret sharing for pMPL to bridge the gap between vector space secret sharing and machine learning. Finally, the experimental results show that the performance of pMPL is promising when we compare it with the state-of-the-art MPL frameworks. Especially, in the LAN setting, pMPL is around 16× and 5× faster than TF-encrypted (with ABY3 as the back-end framework) for the linear regression, and logistic regression, respectively. Besides, the accuracy of trained models of linear regression, logistic regression, and BP neural networks can reach around 97%, 99%, and 96% on MNIST dataset respectively.

CCS CONCEPTS

• **Security and privacy**;

KEYWORDS

Secure Multi-party Computation, Vector Space Secret Sharing, Privacy-preserving Machine Learning, Robustness

ACM Reference Format:
Lushan Song, Jiaxuan Wang, Zhexuan Wang, Xinyu Tu, Guopeng Lin, Wenqiang Ruan, Haoqi Wu, and Weili Han . 2022. pMPL: A Robust Multi-Party Learning Framework with a Privileged Party. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3548606.3560697

1 INTRODUCTION

Privacy-preserving machine learning based on secure multi-party computation (MPC for short), referred to as secure multi-party learning (MPL for short) [32], allows multiple parties to jointly perform machine learning over their private data while protecting the privacy of the raw data. MPL breaks the barriers that different organizations or companies cannot directly share their private raw data mainly due to released privacy protection regulations and laws [30] (e.g. GDPR [33]). Therefore, MPL can be applied to several practical fields involving private data, such as risk control in the financial field [8] and medical diagnosis [13, 14].

Researchers have proposed a doze of MPL frameworks [6, 7, 10, 20, 24, 26, 34], which support ≥2 computation parties during the learning. The involved parties usually follow the peer-to-peer architecture according to the protocols that they rely on. That is, each of them has the same chance to handle the results, including intermediate results and the final model after training. In ABY3 [24], for example, any two parties can cooperate with each other to obtain the final model after training. However, it is also necessary to provide a hierarchical architecture, where a party has its privileged position to handle the process and results of learning due to its motivation and possible payments (including computing resources, and money), in practical scenarios.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560697

1.1 Practical Scenarios

As is shown in Figure 1, three parties, i.e. FinTech, P_1 and P_2, are involved in a scenario of the financial risk control: FinTech is a professional company (usually with a big volume of authorized data and capital) in the financial field. While P_1 and P_2 are two Internet service providers, which usually have lots of valued data (with the authorization from their users). FinTech wants to cooperate with P_1 and P_2 to train an accurate model for the financial risk control, under the payments for the data, which are used in the training process, from P_1 and P_2. However, FinTech, P_1 and P_2 cannot exchange the raw data with each other due to the released privacy protection regulations and laws (e.g. GDPR [33]). Besides, one party could suffer system or network failures, or intentionally quit the training process of machine learning for business purposes, e.g. requiring more payments. Thus, the proposed framework should tolerate the dropping out of a party (P_1 or P_2). For the former case, although parties could restart the training process to deal with the dropping, it should be more practical that the training process is continued to the end, because it can ensure the scheduled deadlines and/or save used computing resources. For the latter case, the proposed framework must support continuing the joint secure training only with the rest parties.

In the above scenario, FinTech requires a privileged position under the payments: (1) FinTech is the only party to reveal the final model, even when P_1 and P_2 collude with each other; (2) After being launched, the training process can be continued to the end, even when P_1 or P_2 drops out due to objective or subjective reasons. Note that FinTech can leverage the robustness to choose one party to reveal the final model, thus keeping its privileged position until the end of training. With the privileged position, FinTech will be much motivated and responsible to deploy MPL frameworks among parties. Thus, the hierarchical architecture is necessary for the development of the studies of MPL frameworks.

As is shown in Figure 1, three parties, i.e. FinTech, P_1 and P_2, hold shares rather than raw data to train models with the support of a series of MPC protocols. After the training, P_1 and P_2 send their shares of the trained model to FinTech to ensure that FinTech is the sole one to reveal the final model. Note that P_1 and P_2 cannot reveal the final model even by colluding with each other. Furthermore, for the second requirement, after three parties hold shares, the training process can be continued with shares of FinTech+ P_1 or FinTech+ P_2 if P_2 or P_1 drops out.

1.2 Related Work

Privacy-preserving machine learning, especially based on MPC technologies, has become a hot spot in recent years. Researchers have proposed a doze of MPL frameworks [6, 7, 10, 20, 24, 26, 34].

Several MPL frameworks were designed based on additive secret sharing [3]. For instance, Mohassel and Zhang [26] proposed a two-party MPL framework, referred to as SecureML, which supported the training of various machine learning models, including linear regression, logistic regression, and neural networks. Wagh et al. [34] designed a three-party MPL framework SecureNN based on additive secret sharing. They eliminated expensive cryptographic operations for the training and inference of neural networks. In the above MPL frameworks, the training would be aborted if one party dropped out.

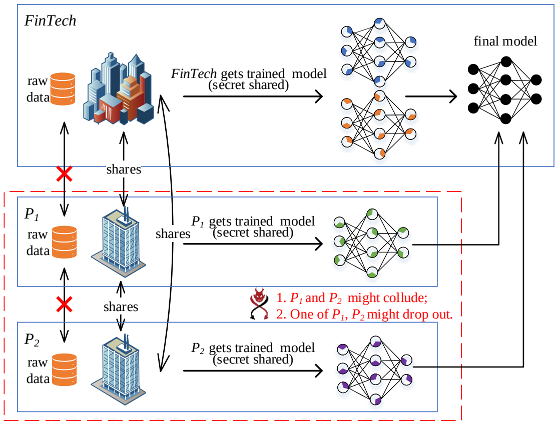
FinTech
raw data
FinTech gets trained model (secret shared)
final model
shares
P1
raw data
shares
P1 gets trained model (secret shared)
1. P1 and P2 might collude;
2. One of P1, P2 might drop out.
shares
P2
raw data
P2 gets trained model (secret shared)

Figure 1: Practical scenarios

In addition, a majority of MPL frameworks were designed based on replicated secret sharing [1]. Mohassel and Rindal [24] proposed ABY3, a three-party MPL framework. It supported efficiently switching back and forth among arithmetic sharing [3], binary sharing [17], and Yao sharing [25]. Trident [7] extended ABY3 to four-party scenarios, and outperformed it in terms of the communication complexity. In both ABY3 and Trident, any two parties can corporate to reveal the secret value (e.g. the final model after training). Therefore, ABY3 and Trident can ensure the robustness that tolerated one of the parties dropping out in the semi-honest security model. Furthermore, several MPL frameworks [6, 10, 20] were designed to tolerate the dropping out of one malicious party during training. That is, even though there existed a malicious party, these MPL frameworks can still continue training, and produce correct outputs. FLASH [6] and SWIFT [20] assumed that there existed one malicious party and three honest parties. They ensured robustness by finding an honest party among four parties, and delegating the training to it. Fantastic Four [10] assumed there existed one malicious party and three semi-honest parties. It ensured the robustness by excluding the malicious party, and the rest parties can continue training securely. Note that the approaches of FLASH and SWIFT would leak the sensitive information of other parties to the honest party, while Fantastic Four would not leak the sensitive information during training. However, any two parties of Fantastic Four (including FLASH and SWIFT) can corporate to reveal the final results. In summary, Fantastic Four cannot set a *privileged party* because it followed a peer-to-peer architecture.

The existing MPL frameworks [6, 7, 10, 20, 24, 26, 34] cannot meet both two requirements mentioned above, although these two ones are important in practical scenarios. For MPL frameworks [26, 34] based on additive secret sharing, they can only meet the first requirement, while cannot meet the second one because when one of the *assistant parties* drops out during training, the machine learning tasks will be aborted. At the same time, several MPL frameworks [6, 7, 10, 20, 24] based on replicated secret sharing have such robustness in the second requirement, while cannot meet the first one, because the final results can be revealed by the cooperation of any t (≤n) parties. That is, these frameworks follow the peer-to-peer architecture.

In addition to MPL, federated learning [18, 19, 36] and trusted execution environments [28] are two other paradigms of privacy-preserving machine learning. In federated learning, each client trains a model with its owned data locally, and uploads the model updates rather than the raw data to a centralized server. Although federated learning has a relatively higher efficiency than that of MPL frameworks, the model updates might contain sensitive information, which might be leaked [23, 39] to the server and other involved clients. In addition, in federated learning, Shamir's secret sharing [31] can be used to ensure the robustness that tolerates part of clients dropping out during the training [4]. The differences between federated learning and our proposed framework will be discussed in Section 6.4. For trusted execution environments, they train models over a centralized data source from distributed locations based on extra trusted hardware. The security model has one or several third trusted parties, thus significantly differs from those of MPL frameworks. The privacy is preserved by the trustworthiness of the data process environment, where parties only obtain the final results without knowing the details of raw data.

1.3 Our Contributions

In this paper, we are motivated to leverage the vector space secret sharing [5], which is typically applied in the cryptographic access control field, to meet the above requirements. Based on vector space secret sharing, we propose a robust MPL framework with a *privileged party*, referred to as pMPL[1]. Given an access structure on a set of parties, the vector space secret sharing guarantees that only the parties in the preset authorized sets can reveal the secret value shared between/among parties. Thus, by setting each authorized set to include the *privileged party* mentioned above, pMPL can meet the first requirement. To ensure the robustness mentioned in the second requirement, we let the *privileged party* hold redundant shares to continue the machine learning when one *assistant party* drops out. Despite the above configuration, how to apply the vector space secret sharing to machine learning, including the technical issues of framework design, efficient protocols, and performance optimizations, is still highly challenging.

We highlight the main contributions in our proposed pMPL as follows:

- **A robust three-party learning framework with a *privileged party*.** We propose pMPL, a three-party learning framework based on vector space secret sharing with a privileged party. pMPL guarantees that only the *privileged party* can obtain the final model even when two *assistant parties* collude with each other. Meanwhile, pMPL is robust, i.e. it can tolerate either of the *assistant parties* dropping out during training. To the best of our knowledge, pMPL is the first framework of privacy-preserving machine learning based on vector space secret sharing.
- **Vector space secret sharing based protocols for pMPL.** Based on the vector space secret sharing, we propose several fundamental efficient protocols required by machine learning in pMPL, including secure addition, secure multiplication, secure conversion between vector space secret sharing and additive secret sharing, secure truncation. Furthermore, to efficiently execute secure multiplication, we design the vector multiplication triplet generation protocol in the offline phase.

Optimized Implementation: Our framework pMPL can be used to train various typical machine learning models, including linear regression, logistic regression, and BP neural networks. We evaluate pMPL on MNIST dataset. The experimental results show that the performance of pMPL is promising compared with the state-of-the-art MPL frameworks, including SecureML and TF-Encrypted [9] (with ABY3 [24] as the back-end framework). Especially, in the LAN setting, pMPL is around 16× and 5× faster than TF-encrypted for the linear regression and logistic regression, respectively. In the WAN setting, although pMPL is slower than both SecureML and TF-encrypted, the performance is still promising. In pMPL, to provide more security guarantees (i.e., defending the collusion of two *assistant parties*) and ensure robustness, pMPL requires more communication overhead. Besides, the accuracy of trained models of linear regression, logistic regression, and BP neural networks can reach around 97%, 99%, and 96% on MNIST dataset, respectively. Note that the accuracy evaluation experiments of linear regression and logistic regression execute the binary classification task, while the evaluation experiments of BP neural networks execute the ten-class classification task.

2 PRELIMINARIES

In this section, we introduce the background knowledge of MPC technologies and three classical machine learning models supported by pMPL.

2.1 Secure Multi-Party Computation

MPC provides rigorous security guarantees and enables multiple parties, which could be mutually distrusted, to cooperatively compute a function while keeping the privacy of the input data. It was firstly introduced by Andrew C. Yao in 1982, and originated from the millionaires' problem [37]. After that, MPC is extended into a general definition for securely computing any function with polynomial time complexity [38]. Various MPC protocols, such as homomorphic encryption-based protocols [16], garbled circuit-based protocols [29], and secret sharing-based protocols [3] have their specific characteristics, and are suitable for different scenarios.

Secret sharing, which typically works over integer rings or prime fields, has proven its feasibility and efficiency in privacy-preserving machine learning frameworks [6, 20, 34]. These frameworks are essentially built on additive secret sharing or replicated secret sharing [1], where the secret value for sharing is randomly split into several shares, the sum of these shares is equal to the secret value. Shamir's secret sharing [31] is another important branch of secret sharing. In Shamir's secret sharing, the shares are constructed according to a randomized polynomial, and the secret value can be reconstructed by solving this polynomial with Lagrange interpolation.

According to the brief analysis of the two requirements of pMPL in Section 1, neither two types of secret sharing mentioned above can meet the both requirements, i.e. supporting a *privileged party* and tolerating that part of *assistant parties* dropping out. Therefore, in our proposed pMPL, we employ the vector space secret sharing [5], another type of secret sharing, to meet the both two requirements.

[1] We open our implementation codes at GitHub (https://github.com/FudanMPL/pMPL).

2.2 Vector Space Secret Sharing

Vector space secret sharing [5] can set which parties can cooperate to reveal the secret value, and which parties cannot reveal the secret value even if they collude with each other.

Let $\mathcal{P} = \{P_0, P_1, \ldots, P_n\}$ be a set of parties (P_i refers to the *i-th* party), and $\Gamma = \{B_0, B_1, \ldots, B_k\}$ be a set of subsets of $\mathcal{P}$, i.e. $\Gamma \subseteq 2^{\mathcal{P}}$. Γ is defined as an access structure on $\mathcal{P}$. Meanwhile, its element $B_j \in \Gamma$ is defined as authorized sets in which parties can cooperate with each other to reveal the secret value. In contrast, the set of parties that is not in the access structure Γ cannot reveal the secret value. Then, with a large prime number p and an integer d where $d \geq 2$, we notify $(\mathbb{Z}_p)^d$ as the vector space over $\mathbb{Z}_p$. Suppose there is a function $\Phi : \mathcal{P} \longrightarrow (\mathbb{Z}_p)^d$ that satisfies the following property:

$$\begin{aligned}&(1,0,\ldots,0) \textit{ can be written as a linear combination of}\\ &\textit{elements in the set } \{\Phi(P_i) \mid P_i \in B_j\} \Leftrightarrow B_j \in \Gamma\end{aligned} \tag{1}$$

That is, for any authorized set B_j, $(1,0,\ldots,0)$ can be represented linearly by all the *public vectors* in the set $\{\Phi(P_i) \mid P_i \in B_j\}$. Therefore, there are m public constants $c_0, ..., c_{m-1}$ (we name them as reconstruction coefficients in this paper), where m refers to the number of parties in B_j, such that:

$$(1,0,\ldots,0) = \sum_{P_i \in B_j} c_i \cdot \Phi(P_i) \tag{2}$$

We denote the matrix constructed by the *public vectors* as $\Phi(\mathcal{P})$, and name it the *public matrix*. Suppose that the *public matrix* $\Phi(\mathcal{P})$ has been determined by all the parties. To secret share a value x, the party who holds this value samples $d-1$ random values $s_1, s_2, \ldots, s_{d-1} \in \mathbb{Z}_p$. Then it constructs the vector $\vec{s} = (x, s_1, s_2, \ldots, s_{d-1})^T$. After that, this party computes the share $x_i = \Phi(P_i) \times \vec{s}$ corresponding to P_i, where $0 \leq i \leq n$.

According to the above share generation mechanism, we can observe that $(1,0,\ldots,0) \times \vec{s} = x$. Hence:

$$x = \Big(\sum_{P_i \in B_j} c_i \cdot \Phi(P_i)\Big) \times \vec{s} = \sum_{P_i \in B_j} c_i \cdot x_i, B_j \in \Gamma \tag{3}$$

Therefore, parties can reveal the secret value x by computing Equation (3).

2.3 Machine Learning Models

We introduce three typical machine learning models supported by pMPL as follows:

Linear Regression: With a matrix of training samples X and the corresponding vector of label values Y, linear regression learns a function G, such that $G(\mathrm{X}) = \mathrm{X} \times \vec{w} \approx \mathrm{Y}$, where $\vec{w}$ is a vector of coefficient parameters. The goal of linear regression is to find the coefficient vector $\vec{w}$ that minimizes the difference between the output of function G and label values. The forward propagation stage in linear aggression is to compute $\mathrm{X} \times \vec{w}$. Then, in the backward propagation stage, the coefficient parameters $\vec{w}$ can be updated as :

$$\vec{w} := \vec{w} - \alpha \mathrm{X}^T(\mathrm{X} \times \vec{w} - \mathrm{Y}) \tag{4}$$

where α is the learning rate.

Logistic Regression: In binary classification problems, logistic regression introduces the logistic function $f(u) = \frac{1}{1+e^{-u}}$ to bound the output of the prediction between 0 and 1. Thus the relationship of logistic regression is expressed as $G(\mathrm{X}) = f(\mathrm{X} \times \vec{w})$. The forward propagation stage in logistic regression is to compute $f(\mathrm{X} \times \vec{w})$. Then, in the backward propagation stage, the coefficient parameters $\vec{w}$ can be updated as:

$$\vec{w} := \vec{w} - \alpha \mathrm{X}^T(f(\mathrm{X} \times \vec{w}) - \mathrm{Y}) \tag{5}$$

BP Neural Networks: Back propagation (BP for short) neural networks can learn non-linear relationships among high dimensional data. A typical BP neural network consists of one input layer, one output layer, and multiple hidden layers. Each layer contains multiple nodes, which are called neurons. Except for the neurons in the input layer, each neuron in other layers comprises a linear function, followed by a non-linear activation function $f(u)$ (e.g. ReLu). In addition, neurons in the input layer take training samples as the input, while other neurons receive their inputs from the previous layer, and process them to produce the computing results that serve as the input to the next layer.

We denote the input matrix as X_0, the coefficient matrix of the $(i-1)$-*th* layer to the *i-th* layer as W_i and the output matrix as Y_m. In the forward propagation stage in BP neural networks, the output of the i-th layer is computed as $\mathrm{A}_i = f(U_i)$, where $\mathrm{U}_i = \mathrm{A}_{\mathrm{i}-1} \times \mathrm{W}_i$, and $f(\cdot)$ is the activation function of the i-th layer. In addition, A_0 is initialized as X_0, and the output matrix is A_m. In the backward propagation stage, the error matrix for the output layer is computed as $\mathrm{E}_m = (\mathrm{A}_m - \mathrm{Y}_m)$, and the error matrices of other layers are computed as $\mathrm{E}_i = (\mathrm{E}_{i+1} \times \mathrm{W}_i^T) \odot \partial f(\mathrm{U}_i)$. Here $\odot$ denotes the element-wise product, and $\partial f(\cdot)$ denotes the derivative of activation function $f(\cdot)$. After the backward propagation phase, we update the coefficient matrix as $\mathrm{W}_i := \mathrm{W}_i - \alpha \mathrm{A}_{i-1}^T \times \mathrm{E}_i$.

3 OVERVIEW OF pMPL

In this section, we firstly describe the architecture of pMPL, and introduce the data representation of pMPL. After that, we present the security model considered in this paper. Finally, we introduce the design of robust training of pMPL. For the clarity purpose, we show the notations used in this paper in Table 1.

Table 1: Notations used in this paper.

Symbol	Description
$\mathcal{P}$	The set of parties
Γ	The access structure
B_j	The authorized set
$[\cdot]$	The shares of additive secret sharing
$[\cdot]^2$	The shares of boolean sharing
$\langle\cdot\rangle$	The shares of vector space secret sharing
$\Phi(\mathcal{P})$	The *public matrix* for vector space secret sharing
$c_0, c_1, \ldots, c_3''$	The reconstruction coefficients
a_0, a_1	The coefficients of the alternate vector
ℓ	The number of bits to represent a fixed-point number
ℓ_f	The number of bits to represent the fractional part of a fixed-point number
$\langle u\rangle, \langle v\rangle, \langle h\rangle$	The vector multiplication triplet
B	The batch size
D	The dimension of the feature
E	The number of the epoch

3.1 Architecture and Data Representation

3.1.1 Architecture. As is shown in Figure 2, we consider a set of three parties $\mathcal{P} = \{P_0, P_1, P_2\}$, who want to train various machine learning models over their private raw data jointly. Without loss of generality, we define P_0 as the *privileged party* and P_1, P_2 as *assistant parties*. These parties are connected by secure pairwise communication channels in a synchronous network. Before training, these parties secret share (using the $\langle\cdot\rangle$-*sharing* semantics introduced in Section 4.1) their private raw data with each other. During training, all the parties communicate the shared form $\langle Msg\rangle$ of intermediate messages with each other. In pMPL, the *privileged party* P_0 holds $\langle Msg\rangle_0$ and $\langle Msg\rangle_3$, and *assistant parties* P_1 and P_2 hold $\langle Msg\rangle_1$ and $\langle Msg\rangle_2$ respectively. During the training process, none of the parties can get others' raw data or infer any private information from the intermediate results and the final model.

Besides, the final model is supposed to be obtained only by *privileged party* P_0. Furthermore, pMPL tolerates one *assistant party* (P_1 or P_2) dropping out of training. As a result, the access structure Γ in pMPL is $\{\{P_0, P_1, P_2\}, \{P_0, P_1\}, \{P_0, P_2\}\}$.

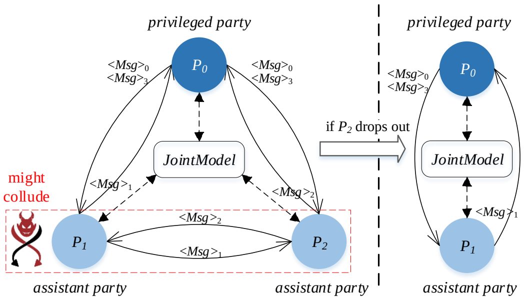

Figure 2: Overview of pMPL

3.1.2 Data representation. In machine learning, to train accurate models, most of the intermediate values are represented as floating-point numbers. However, since the precision of floating-point numbers is not fixed, every calculation requires additional operations for alignment. Therefore, floating-point calculations would lead to more computation and communication overhead.

In order to balance the accuracy and efficiency of the floating-point calculations in pMPL, we handle floating-point values with a fixed-point representation. More specifically, we denote a fixed-point decimal as an ℓ-bit integer, which is identical to the previous MPL frameworks (e.g. SecureML [26]). Among these ℓ bits, the most significant bit (MSB) represents the sign and the ℓ_f least significant bits are allocated to represent the fractional part. An ℓ-bit integer can be treated as an element of a ring $\mathbb{Z}_{2^\ell}$. Note that to ensure that corresponding reconstruction coefficients can be computed for any *public matrix*, vector space secret sharing usually performs on a prime field. However, it is more efficient to work on a ring [11]. Therefore, we perform our computations on a ring $\mathbb{Z}_{2^\ell}$ by restricting the *public matrix* (see Section 4.2 for more detail).

3.2 Security Model

In this paper, we employ the semi-honest (also known as honest-but-curious or passive) security model in pMPL. A semi-honest adversary attempts to infer as much information as possible from the messages they received during training. However, they follow the protocol specification. Furthermore, we have an asymmetric security assumption that *assistant parties* P_1 and P_2 might collude, and the *privileged party* P_0 would not collude with any *assistant party*. This setting is different from those of the previous MPL frameworks (e.g. SecureML [26] and ABY3 [24]).

3.3 Robust Training

The robustness employed in pMPL ensures that training would continue even though one *assistant party* drops out. In pMPL, an additional vector, referred to as the alternate vector, is held by the *privileged party*. The alternate vector can be represented linearly by the vectors held by two *assistant parties*. Here, we denote all shares generated by the alternate vector as alternate shares. During training, if no *assistant party* drops out, these alternate shares are executed with the same operations as other shares. Once one *assistant party* drops out, the alternate shares would replace the shares held by the dropped party. Thus the rest two parties can continue training.

With the robustness, the *privileged party* can tolerate the dropping out of one *assistant party*, even though the *assistant party* intentionally quit the training process. Furthermore, the *privileged party* can choose one *assistant party* to reveal the final model, thus keeping its privileged position until the end of the training.

4 DESIGN OF PMPL

In this section, we firstly introduce the sharing semantics of pMPL, as well as sharing and reconstruction protocols. After that, we show the basic primitives and the building blocks that are designed to support 3PC training in pMPL. Furthermore, we introduce the design of robustness of pMPL. Finally, we analyze the complexity of our proposed protocols.

4.1 Sharing Semantics

In this paper, we leverage two types of secret sharing protocols, $\langle\cdot\rangle$-sharing and $[\cdot]$-sharing:

- $\langle\cdot\rangle$-sharing: We use $\langle\cdot\rangle$ to denote the shares of vector space secret sharing. The more detailed descriptions of sharing protocol and reconstruction protocol are shown in Section 4.2.
- $[\cdot]$-sharing: We use $[\cdot]$ to denote the shares of additive secret sharing. A value $x \in \mathbb{Z}_{2^\ell}$ is said to be $[\cdot]$-shared among a set of parties $\mathcal{P} = \{P_0, P_1, P_2\}$, if each party P_i holds $[x]_i \in \mathbb{Z}_{2^\ell}$ ($i \in \{0,1,2\}$), such that $x = ([x]_0 + [x]_1 + [x]_2)\ mod\ 2^\ell$, which is represented as $x = [x]_0 + [x]_1 + [x]_2$ in the rest of the paper. Besides, we define the boolean sharing as $[\cdot]^2$, which refers to the shares over $\mathbb{Z}_2$.

Note that we use $\langle\cdot\rangle$-sharing as the underlying technique of pMPL. Besides, $[\cdot]$-sharing is only used for the comparison protocol to represent the intermediate computation results.

Linearity of the Secret Sharing Schemes: Given the $\langle\cdot\rangle$-sharing of x, y and public constants k_1, k_2, each party can locally compute $\langle k_1 \cdot x + k_2 \cdot y\rangle = k_1 \cdot \langle x\rangle + k_2 \cdot \langle y\rangle$. Besides, it is obvious that $[\cdot]$-sharing also satisfies the linearity property. The linearity property enables parties to non-interactively execute addition operations, as well as execute multiplication operations of their shares with a public constant.

4.2 Sharing and Reconstruction Protocols

In pMPL, to share a secret value x, we form it as a three-dimensional vector $\vec{s} = (x, s_1, s_2)^T$, where s_1 and s_2 are two random values. We define a *public matrix* $\Phi(\mathcal{P})$ as a 4×3 matrix. Here, for each party P_i, the *i-th* row $\Phi(i)$ of $\Phi(\mathcal{P})$ is its corresponding three-dimensional *public vector*. Besides, the *privileged party* P_0 holds the alternate three-dimensional *public vector* $\Phi(3)$.

To meet the two requirements mentioned in Section 1.1, the *public matrix* $\Phi(\mathcal{P})$ should satisfy four restrictions as follows:

- $(1,0,0)$ can be written as a linear combination of the *public vectors* in the set $\{\Phi(0),\Phi(1),\Phi(2)\}$, where $\Phi(0),\Phi(1),\Phi(2)$ are linearly independent. Thus there are three non-zero public constants c_0, c_1, c_2, such that $(1,0,0) = c_0 \cdot \Phi(0) + c_1 \cdot \Phi(1) + c_2 \cdot \Phi(2)$.
- The *public vector* $\Phi(3)$ can be represented linearly by the vectors $\Phi(1)$ and $\Phi(2)$, i.e. $\Phi(3) = a_1 \cdot \Phi(1) + a_2 \cdot \Phi(2)$, where $a_1, a_2 \neq 0$. Therefore, $(1,0,0)$ can also be written as a linear combination of the *public vectors* in both sets $\{\Phi(0),\Phi(1),\Phi(3)\}$ and $\{\Phi(0),\Phi(2),\Phi(3)\}$. That is, there are six non-zero public constants $c'_0, c'_1, c'_3, c''_0, c''_2, c''_3$, such that $(1,0,0) = c'_0 \cdot \Phi(0) + c'_1 \cdot \Phi(1) + c'_3 \cdot \Phi(3) = c''_0 \cdot \Phi(0) + c''_2 \cdot \Phi(2) + c''_3 \cdot \Phi(3)$.
- To prevent the set of parties that are not in the access structure from revealing the secret value, $(1,0,0)$ cannot be written as a linear combination of the *public vectors* in both the sets $\{\Phi(0),\Phi(3)\}$ and $\{\Phi(1),\Phi(2)\}$.
- As pMPL performs the computations on the ring $\mathbb{Z}_{2^\ell}$, both the values of *public matrix* $\Phi(\mathcal{P})$ and reconstruction coefficients $c_0, c_1, \ldots, c''_3$ should be elements of the ring $\mathbb{Z}_{2^\ell}$.

We formalize the above restrictions as Equation (6) as follows:

$$\begin{aligned}(1,0,0) &= c_0 \cdot \Phi(0) + c_1 \cdot \Phi(1) + c_2 \cdot \Phi(2)\\ &= c'_0 \cdot \Phi(0) + c'_1 \cdot \Phi(1) + c'_3 \cdot \Phi(3)\\ &= c''_0 \cdot \Phi(0) + c''_2 \cdot \Phi(2) + c''_3 \cdot \Phi(3)\end{aligned} \tag{6}$$

Once the *public matrix* $\Phi(\mathcal{P})$ is determined, the reconstruction coefficients $c_0, c_1, \ldots, c''_3$ can be computed by Equation (6). It is trivial that these coefficients are also public to all parties.

Protocol 1 $\prod_{\text{shr}}(P_i, x)$

```
Input: The secret value $x$ held by $P_i$
Output: $\langle x \rangle$
1: $P_i$ constructs a three-dimensional vector $\vec{s} = (x, s_1, s_2)^T$, where $s_1$ and $s_2$ are random values.
2: - If $P_i = P_0$, $P_i$ sends $\langle x \rangle_j = \Phi(j) \times \vec{s}$ to $P_j$ for $j \in \{1,2\}$. Meanwhile, $P_i$ generates $\langle x \rangle_0 = \Phi(0) \times \vec{s}$ and $\langle x \rangle_3 = \Phi(3) \times \vec{s}$ for itself.
   - If $P_i \neq P_0$, $P_i$ sends $\langle x \rangle_j = \Phi(j) \times \vec{s}$ to $P_j$ for $j \in \{0,1,2\}\backslash\{i\}$, and sends the alternate share $\langle x \rangle_3 = \Phi(3) \times \vec{s}$ to $P_0$. Meanwhile, $P_i$ generates share $\langle x \rangle_i = \Phi(i) \times \vec{s}$ for itself.
```

Sharing Protocol: As is shown in Protocol 1, $\prod_{\text{shr}}(P_i, x)$ enables P_i who holds the secret value x to generate $\langle\cdot\rangle$-shares of x. In Step 1 of $\prod_{\text{shr}}(P_i, x)$ (Protocol 1), P_i samples two random values s_1 and s_2 to construct a three-dimensional vector $\vec{s} = (x, s_1, s_2)^T$. In Step 2 of $\prod_{\text{shr}}(P_i, x)$ (Protocol 1), we consider two cases as follows: (1) If $P_i = P_0$, P_i sends $\langle x \rangle_j = \Phi(j) \times \vec{s}$ to two *assistant parties* P_j for $j \in \{1,2\}$. Meanwhile, P_i generates $\langle x \rangle_0 = \Phi(0) \times \vec{s}$ as well as the alternate share $\langle x \rangle_3 = \Phi(3) \times \vec{s}$, and holds them. (2) If $P_i \neq P_0$, P_i sends $\langle x \rangle_j = \Phi(j) \times \vec{s}$ to P_j for $j \in \{0,1,2\}\backslash\{i\}$. Besides, P_i sends the alternate share $\langle x \rangle_3 = \Phi(3) \times \vec{s}$ to P_0 and holds $\langle x \rangle_i = \Phi(i) \times \vec{s}$. After the execution of $\prod_{\text{shr}}(P_i, x)$ (Protocol 1), P_0 holds $\langle x \rangle_0$ and $\langle x \rangle_3$, P_1 holds $\langle x \rangle_1$, and P_2 holds $\langle x \rangle_2$. We use the standard real/ideal world paradigm to prove the security of $\prod_{\text{shr}}(P_i, x)$ in Appendix B.

Reconstruction Protocol: According to Equation (6) and $\prod_{\text{shr}}(P_i, x)$ (Protocol 1), we can reveal the secret value x through Equation (7), (8), or (9) for different scenarios:

$$x = c_0 \cdot \langle x \rangle_0 + c_1 \cdot \langle x \rangle_1 + c_2 \cdot \langle x \rangle_2 \tag{7}$$

$$= c'_0 \cdot \langle x \rangle_0 + c'_1 \cdot \langle x \rangle_1 + c'_3 \cdot \langle x \rangle_3 \tag{8}$$

$$= c''_0 \cdot \langle x \rangle_0 + c''_2 \cdot \langle x \rangle_2 + c''_3 \cdot \langle x \rangle_3 \tag{9}$$

As is shown in Protocol 2, $\prod_{\text{rec}}(\mathcal{P}, \langle x \rangle)$ enables parties to reveal the secret value x. Without loss of generality, we assign P_2 as the dropping *assistant party* when one party drops out, as is shown in Figure 2. We consider two cases as follows: (1) If no *assistant party* drops out, each party P_i receives shares from the other two parties. Then they compute Equation (7) to reveal the secret value x (P_i can also reveal the secret value x by computing Equation (8) or (9).). (2) If P_2 drops out, P_0 receives the shares $\langle x \rangle_1$ from P_1. Meanwhile, P_1 receives the share $\langle x \rangle_0$ and $\langle x \rangle_3$ from P_0. Then P_0 and P_1 non-interactively compute Equation (8) to reveal the secret value x locally. Note that even though P_1 and P_2 collude with each other, without the participation of P_0, the secret value x cannot be revealed in $\prod_{\text{rec}}(\mathcal{P}, \langle x \rangle)$ (Protocol 2). Besides, once training is completed, P_1 and P_2 send their shares to P_0, while P_0 does not send its final shares to other parties. Therefore, only P_0 can obtain the final model. Besides, we use the standard real/ideal world paradigm to prove the security of $\prod_{\text{rec}}(\mathcal{P}, \langle x \rangle)$ in Appendix B.

Protocol 2 $\prod_{\text{rec}}(\mathcal{P}, \langle x \rangle)$

```
Input: $\langle x \rangle$
Output: $x$
- If no party drops out:
  1: $P_i$ receives shares from the other two parties.
  2: $P_i$ reveal $x$ by computing Equations (7): $x = c_0 \cdot \langle x \rangle_0 + c_1 \cdot \langle x \rangle_1 + c_2 \cdot \langle x \rangle_2$.
- If $P_2$ drops out:
  1: $P_0$ receives $\langle x \rangle_1$ from $P_1$. Meanwhile, $P_1$ receives $\langle x \rangle_0$ and $\langle x \rangle_3$ from $P_0$.
  2: $P_0$ and $P_1$ reveal $x$ by computing Equations (8): $x = c'_0 \cdot \langle x \rangle_0 + c'_1 \cdot \langle x \rangle_1 + c'_3 \cdot \langle x \rangle_3$.
```

4.3 Basic Primitives for 3PC

In this section, we introduce the design of the basic primitives in pMPL for 3PC (i.e. no party drops out) in detail, including: (1) the primitives of secure addition and secure multiplication; (2) the primitives of sharing conversion: $\langle\cdot\rangle$-sharing to $[\cdot]$-sharing and $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing; (3) MSB extraction and Bit2A, i.e. boolean to additive conversion. Besides, we use the standard real/ideal world paradigm to prove the security of these basic primitives in Appendix B.

Secure Addition: Given two secret values x and y, each party P_i holds shares $\langle x \rangle_i$ and $\langle y \rangle_i$ (P_0 additionally holds the alternate shares $\langle x \rangle_3$ and $\langle y \rangle_3$). To get the result of secure addition $\langle x + y \rangle$, each party P_i can utilize the linearity property of the $\langle\cdot\rangle$-sharing scheme to locally compute $\langle z \rangle_i = \langle x \rangle_i + \langle y \rangle_i$. P_0 additionally computes $\langle z \rangle_3 = \langle x \rangle_3 + \langle y \rangle_3$ for the alternate shares.

Secure Multiplication: Through interactive computing, parties securely multiply two shares $\langle x \rangle$ and $\langle y \rangle$. According to Equation

(10), we utilize two random values u and v to mask the secret values x and y. More specifically, we utilize a vector multiplication triplet (u, v, h), which refers to the method of Beaver's multiplication triplet [2], to execute secure multiplication.

$$\begin{aligned} x \cdot y &= x \cdot (y + v) - x \cdot v = x \cdot (y + v) - v \cdot (x + u - u) \\ &= x \cdot (y + v) - v \cdot (x + u) + v \cdot u \end{aligned} \tag{10}$$

Protocol 3 shows the secure multiplication protocol $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ proposed in pMPL. Besides, the shares held by each party during the execution of secure multiplication, which consists of five steps, are shown in Appendix A.1, (concretely in Table 7). In the offline phase of $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ (Protocol 3), we set $\vec{r} = (u, r_1, r_2)^T$, $\vec{q} = (v, q_1, q_2)^T$ uniformly random three-dimensional vectors and $\vec{t} = (h, t_1, t_2)^T = (u \cdot v, t_1, t_2)^T$, where t_1, t_2 are uniformly random values. We assume that all the parties have already shared vector multiplication triplet ($\langle u \rangle$, $\langle v \rangle$, $\langle h \rangle$) in the offline phase. In the online phase of $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ (Protocol 3), firstly, each party P_i locally computes $\langle e \rangle_i = \langle x \rangle_i + \langle u \rangle_i$ and $\langle f \rangle_i = \langle y \rangle_i + \langle v \rangle_i$. P_0 additionally computes the alternate shares $\langle e \rangle_3 = \langle x \rangle_3 + \langle u \rangle_3$ and $\langle f \rangle_3 = \langle y \rangle_3 + \langle v \rangle_3$ locally. To get e and f, parties then interactively execute $\prod_{\text{rec}}(\mathcal{P}, \langle e \rangle)$ (Protocol 2) and $\prod_{\text{rec}}(\mathcal{P}, \langle f \rangle)$ (Protocol 2). Finally, each party P_i locally computes $\langle z \rangle_i = \langle x \rangle_i \cdot f - \langle v \rangle_i \cdot e + \langle h \rangle_i$. Similarly, P_0 additionally computes the alternate share $\langle z \rangle_3 = \langle x \rangle_3 \cdot f - \langle v \rangle_3 \cdot e + \langle h \rangle_3$.

Protocol 3 $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$

Preprocessing: Parties pre-shared vector multiplication triplet $\langle u \rangle$, $\langle v \rangle$, $\langle h \rangle$ using $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4)
Input: $\langle x \rangle$ and $\langle y \rangle$
Output: $\langle x \cdot y \rangle$

1: P_i locally computes $\langle e \rangle_i = \langle x \rangle_i + \langle u \rangle_i$ and $\langle f \rangle_i = \langle y \rangle_i + \langle v \rangle_i$. P_0 additionally computes $\langle e \rangle_3 = \langle x \rangle_3 + \langle u \rangle_3$ and $\langle f \rangle_3 = \langle y \rangle_3 + \langle v \rangle_3$.
2: Parties interactively execute $\prod_{\text{rec}}(\mathcal{P}, \langle e \rangle)$ (Protocol 2) and $\prod_{\text{rec}}(\mathcal{P}, \langle f \rangle)$ (Protocol 2).
3: P_i locally computes $\langle z \rangle_i = \langle x \rangle_i \cdot f - \langle v \rangle_i \cdot e + \langle h \rangle_i$ and P_0 additionally computes the alternate share $\langle z \rangle_3 = \langle x \rangle_3 \cdot f - \langle v \rangle_3 \cdot e + \langle h \rangle_3$.

The vector multiplication triplets can be generated by a cryptography service provider (CSP) or securely generated by multiparty collaboration. $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4) enables parties to securely generate expected shared vector multiplication triplets ($\langle u \rangle, \langle v \rangle, \langle h \rangle$). It consists of two phases, i.e. generating $\langle u \rangle$, $\langle v \rangle$ and generating $\langle h \rangle$. Moreover, the shares that each party holds during the execution of $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4), which consists of seven steps, are shown in Appendix A.2 (concretely in Table 8).

- *Generating $\langle u \rangle$ and $\langle v \rangle$:* As $\langle u \rangle$ and $\langle v \rangle$ are generated in the same way, we hereby take the generation of $\langle u \rangle$ as an example. Firstly, each party P_i generates a random value u_i. Then they interactively execute $\prod_{\text{shr}}(P_i, u_i)$ (Protocol 1). After that, each party P_i holds three shares $\langle u_0 \rangle_i, \langle u_1 \rangle_i, \langle u_2 \rangle_i$. Besides, P_0 additionally holds another three alternate shares $\langle u_0 \rangle_3, \langle u_1 \rangle_3, \langle u_2 \rangle_3$. Then each party P_i adds up these three shares locally to compute $\langle u \rangle_i = \langle u_0 \rangle_i + \langle u_1 \rangle_i + \langle u_2 \rangle_i$. P_0 additionally computes $\langle u \rangle_3 = \langle u_0 \rangle_3 + \langle u_1 \rangle_3 + \langle u_2 \rangle_3$.
- *Generating $\langle h \rangle$:* Given shared random values $\langle u \rangle$ and $\langle v \rangle$ mentioned above, the key step of generating $\langle h \rangle$ is to compute the shares of their product. According to the process of generating $\langle u \rangle$ and $\langle v \rangle$, we can get that $u = u_0 + u_1 + u_2$ and $v = v_0 + v_1 + v_2$. Then:

$$\begin{aligned} h = uv = (u_0 + u_1 + u_2)(v_0 + v_1 + v_2) &= u_0v_0 + u_0v_1 + u_0v_2 \\ +u_1v_0 + u_1v_1 + u_1v_2 &+ u_2v_0 + u_2v_1 + u_2v_2 \end{aligned} \tag{11}$$

where u_iv_i ($i \in \{0, 1, 2\}$) can be computed locally in each party P_i and the rest products require three parties to compute cooperatively. We use the method proposed by Zhu and Takagi [40] to calculate $[u_0v_1 + u_1v_0]$, $[u_0v_2 + u_2v_0]$, and $[u_1v_2 + u_2v_1]$. After that, each party P_i locally computes $h_i = u_iv_i + [u_iv_{i+1} + u_{i+1}v_i]_i + [u_iv_{i-1} + u_{i-1}v_i]_i$. Here, $i \pm 1$ refers to the next (+) or previous (-) party with wrap around. For example, the party 2 + 1 is the party 0, and the party 0 - 1 is the party 2. Subsequently, each party P_i executes $\prod_{\text{shr}}(P_i, h_i)$ (Protocol 1) to get three shares $\langle h_0 \rangle_i$, $\langle h_1 \rangle_i$ and $\langle h_2 \rangle_i$ (P_0 additionally holds three alternate shares $\langle h_0 \rangle_3$, $\langle h_1 \rangle_3$ and $\langle h_2 \rangle_3$). At last, each party P_i adds up the three shares locally to get $\langle h \rangle_i = \langle h_0 \rangle_i + \langle h_1 \rangle_i + \langle h_2 \rangle_i$ (P_0 additionally adds up three alternate shares to get $\langle h \rangle_3 = \langle h_0 \rangle_3 + \langle h_1 \rangle_3 + \langle h_2 \rangle_3$).

Protocol 4 $\prod_{\text{vmtgen}}(\mathcal{P})$

Input: $\emptyset$
Output: The shares of vector multiplication triplet ($\langle u \rangle$, $\langle v \rangle$, $\langle h \rangle$)
Generating $\langle u \rangle$, $\langle v \rangle$**:**

1: P_i generates two random values u_i and v_i.
2: P_i executes $\prod_{\text{shr}}(P_i, u_i)$ (Protocol 2) and $\prod_{\text{shr}}(P_i, v_i)$ (Protocol 2).
3: P_i locally computes $\langle u \rangle_i = \langle u_0 \rangle_i + \langle u_1 \rangle_i + \langle u_2 \rangle_i$, and $\langle v \rangle_i = \langle v_0 \rangle_i + \langle v_1 \rangle_i + \langle v_2 \rangle_i$. Besides, P_0 computes the alternate shares $\langle u \rangle_3$ and $\langle v \rangle_3$ in the same way.

Generating $\langle h \rangle$**:**

1: P_0 and P_1 interactively compute $[u_0v_1+u_1v_0]$, P_0 and P_2 interactively compute $[u_0v_2+u_2v_0]$, P_1 and P_2 interactively compute $[u_1v_2+u_2v_1]$.
2: P_i locally computes $h_i = u_iv_i + [u_iv_{i+1} + u_{i+1}v_i]_i + [u_iv_{i-1} + u_{i-1}v_i]_i$.
3: P_i executes $\prod_{\text{shr}}(P_i, h_i)$ (Protocol 1).
4: P_i locally computes $\langle h \rangle_i = \langle h_0 \rangle_i + \langle h_1 \rangle_i + \langle h_2 \rangle_i$ and P_0 additionally computes the alternate share $\langle h \rangle_3 = \langle h_0 \rangle_3 + \langle h_1 \rangle_3 + \langle h_2 \rangle_3$.

Sharing Conversion: Previous studies [20][24] have established that non-linear operations such as comparison are more efficient in $\mathbb{Z}_2$ than in $\mathbb{Z}_{2^\ell}$. That is, $[\cdot]^2$-sharing is more suitable for executing non-linear operations than both $\langle \cdot \rangle$-sharing and $[\cdot]$-sharing. However, the conversions between $\langle \cdot \rangle$-shares and $[\cdot]^2$-shares are challenging, while the conversions between $\langle \cdot \rangle$-shares and $[\cdot]$-shares are relatively easy to perform. Thus, to efficiently execute non-linear operations, we firstly convert $\langle \cdot \rangle$-shares to $[\cdot]$-shares locally. Furthermore, we use the existing methods [11][24] to convert between $[\cdot]$-shares and $[\cdot]^2$-shares. Finally, we convert $[\cdot]$-shares back to $\langle \cdot \rangle$-shares.

We hereby present two primitives of sharing conversion as follows:

- *Converting $\langle \cdot \rangle$-shares to $[\cdot]$-shares:* $\prod_{\text{v2a}}(\mathcal{P}, \langle x \rangle)$ enables each party P_i locally computes $[x]_i = c_i \cdot \langle x \rangle_i$ to convert $\langle \cdot \rangle$-shares to $[\cdot]$-shares according to Equation (12).

$$x = c_0 \cdot \langle x \rangle_0 + c_1 \cdot \langle x \rangle_1 + c_2 \cdot \langle x \rangle_2 = [x]_0 + [x]_1 + [x]_2 \tag{12}$$

Here, we only convert three, i.e. $\langle x \rangle_0$, $\langle x \rangle_1$, $\langle x \rangle_2$, of the four $\langle \cdot \rangle$-shares to $[\cdot]$-shares. Since pMPL supports the privileged party and one of two assistant parties (three shares) to train and the reconstruction protocol only needs three shares, this configuration does not affect subsequent operations.

- *Converting* $[\cdot]$*-shares to* $\langle\cdot\rangle$*-shares:* $\prod_{\mathrm{a2v}}(\mathcal{P}, [x])$ (Protocol 5) enables parties to convert $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing. Here, we are supposed to convert three $[\cdot]$-shares to four $\langle\cdot\rangle$-shares. Except for the alternate share, each party P_i locally computes $\langle x\rangle_i = [x]_i/c_i$. Due to the equation: $\Phi(3) = a_1 \cdot \Phi(1) + a_2 \cdot \Phi(2)$, we can get the alternate share $\langle x\rangle_3$ by computing $\langle x\rangle_3 = a_1 \cdot \langle x\rangle_1 + a_2 \cdot \langle x\rangle_2$. We assume that all the parties have already shared a random value k, which is generated in the same way as $\langle u\rangle$ and $\langle v\rangle$ in $\prod_{\mathrm{vmtgen}}(\mathcal{P})$ (Protocol 4). Then P_1 and P_2 compute $\langle x\rangle_j + \langle k\rangle_j$ ($j \in \{1, 2\}$) locally, and send them in plaintext to P_0. Finally, P_0 locally computes the alternate share $\langle x\rangle_3 = a_1 \cdot (\langle x\rangle_1 + \langle k\rangle_1) + a_2 \cdot (\langle x\rangle_2 + \langle k\rangle_2) - \langle k\rangle_3$.

Protocol 5 $\prod_{\mathrm{a2v}}(\mathcal{P}, [x])$

Preprocessing: Parties pre-shared $\langle k\rangle$
Input: $[x]$
Output: $\langle x\rangle$

```
1: $P_i$ locally computes $\langle x\rangle_i = [x]_i/c_i$.
2: $P_1$ and $P_2$ locally compute $\langle x\rangle_j + \langle k\rangle_j$ ($j \in \{1,2\}$) , and send them to $P_0$.
3: $P_0$ locally computes $\langle x\rangle_3 = a_1 \cdot (\langle x\rangle_1 + \langle k\rangle_1) + a_2 \cdot (\langle x\rangle_2 + \langle k\rangle_2) - \langle k\rangle_3$.
```

MSB extraction and Bit2A: The MSB extraction protocol $\prod_{\mathrm{msbext}}(\mathcal{P}, [x])$ enables parties to compute boolean sharing of MSB of a value x (Here, we use the method presented in the study [22], and name it in this paper). Bit2A protocol $\prod_{\mathrm{b2a}}(\mathcal{P}, [b]^2)$ enables parties to compute from the boolean sharing of b ($[b]^2$) to its additive secret sharing ($[b]$) (Here, we use the method presented in the study [11], and name it in this paper).

4.4 Building Blocks for pMPL

We detail the design of the building blocks in pMPL for 3PC as follows: (1) matrix sharing; (2) matrix addition and matrix multiplication; (3) truncation; (4) two activation functions, i.e. ReLU and Sigmoid.

Matrix Sharing: As all the variables in pMPL are represented as matrices. In order to improve the efficiency of sharing protocol, we generalize the sharing operation on a single secret value to an $n \times d$ secret matrix X. As is shown in Figure 3, P_i who holds the secret matrix X firstly flattens X into row vector $\vec{X'}$ with the size of nd. Then P_i constructs a $3 \times nd$ matrix $\mathrm{S}' = (\vec{X'}^T, \vec{S1}^T, \vec{S2}^T)^T$, where $\vec{S1}$ and $\vec{S2}$ are random row vectors with size of nd. Furthermore, P_i computes shares $\langle\vec{X'}\rangle_k = \Phi(k) \times \mathrm{S}'$ for $k = \{0, 1, 2, 3\}$. Finally, P_i converts $\langle\vec{X'}\rangle_k$ to an $n \times d$ matrix $\langle\mathrm{X}\rangle_k$.

Matrix Addition and Multiplication: We generalize the addition and multiplication operations on shares to shared matrices referring to the method of [26]. Given two shared matrices $\langle\mathrm{X}\rangle$ (with the size of $n\times d$) and $\langle\mathrm{Y}\rangle$ (with the size of $d \times m$), in the matrix addition, each party P_i locally computes $\langle\mathrm{Z}\rangle_i = \langle\mathrm{X}\rangle_i + \langle\mathrm{Y}\rangle_i$. P_0 additionally computes the alternate shared matrix $\langle\mathrm{Z}\rangle_3 = \langle\mathrm{X}\rangle_3 + \langle\mathrm{Y}\rangle_3$. To multiply two shared matrices $\langle\mathrm{X}\rangle$ and $\langle\mathrm{Y}\rangle$, instead of using independent vector multiplication triplets (u, v, h) on each element multiplication, we take matrix vector multiplication triplets (U, V, H) to execute the matrix multiplication. Here, U and V are random matrices, U has the same dimension as X, V has the same dimension as Y and $\mathrm{H} = \mathrm{U} \times \mathrm{V}$. We assume that all the parties have already shared ($\langle\mathrm{U}\rangle$, $\langle\mathrm{V}\rangle$, $\langle\mathrm{H}\rangle$). Each party P_i firstly computes $\langle\mathrm{E}\rangle_i = \langle\mathrm{X}\rangle_i + \langle\mathrm{U}\rangle_i$ and $\langle\mathrm{F}\rangle_i = \langle\mathrm{Y}\rangle_i + \langle\mathrm{V}\rangle_i$ locally. P_0 additionally computes $\langle\mathrm{E}\rangle_3 = \langle\mathrm{X}\rangle_3 + \langle\mathrm{U}\rangle_3$ and $\langle\mathrm{F}\rangle_3 = \langle\mathrm{Y}\rangle_3 + \langle\mathrm{V}\rangle_3$. Then parties reveal E and F, and compute $\langle\mathrm{Z}\rangle_i = \langle\mathrm{X}\rangle_i \times \mathrm{F} - \mathrm{E} \times \langle\mathrm{V}\rangle_i + \langle\mathrm{H}\rangle_i$ locally. P_0 additionally computes $\langle\mathrm{Z}\rangle_3 = \langle\mathrm{X}\rangle_3 \times \mathrm{F} - \mathrm{E} \times \langle\mathrm{V}\rangle_3 + \langle\mathrm{H}\rangle_3$.

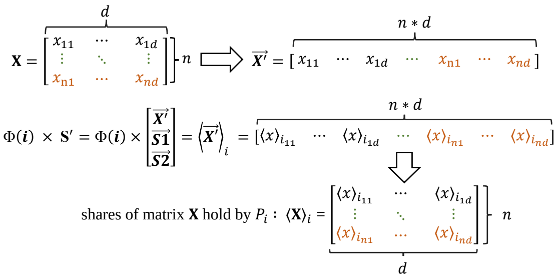
d
$\mathbf{X} =$
x_{11} ⋯ x_{1d}
x_{n1} ⋯ x_{nd}
n
$n * d$
$\vec{X'} = [x_{11}$ ⋯ x_{1d} ⋯ x_{n1} ⋯ $x_{nd}]$
$\Phi(i) \times \mathbf{S}' = \Phi(i) \times [\vec{X'}; \vec{S1}; \vec{S2}] = \langle\vec{X'}\rangle_i = [\langle x\rangle_{i_{11}}$ ⋯ $\langle x\rangle_{i_{1d}}$ ⋯ $\langle x\rangle_{i_{n1}}$ ⋯ $\langle x\rangle_{i_{nd}}]$
shares of matrix $\mathbf{X}$ hold by P_i : $\langle\mathbf{X}\rangle_i =$
$\langle x\rangle_{i_{11}}$ ⋯ $\langle x\rangle_{i_{1d}}$
$\langle x\rangle_{i_{n1}}$ ⋯ $\langle x\rangle_{i_{nd}}$

Figure 3: Matrix conversions during matrix sharing

As for the generation of matrix vector multiplication triplets (U, V, H), the process is similar to $\prod_{\mathrm{vmtgen}}(\mathcal{P})$ (Protocol 4), where the sharing protocol is replaced with the matrix sharing protocol. For the generation of U and V, we also take U as an example. Firstly, each party P_i generates a random $n \times d$ matrix U_i, P_3 additionally generates a random matrix U_3. Then each party P_i shares (using matrix sharing protocol) U_i, P_3 additionally shares matrices U_3. After that, each party P_i holds three shared matrices $\langle\mathrm{U}_0\rangle_i$, $\langle\mathrm{U}_1\rangle_i$, $\langle\mathrm{U}_2\rangle_i$. Besides, P_0 additionally holds another three alternate shares $\langle\mathrm{U}_0\rangle_3$, $\langle\mathrm{U}_1\rangle_3$, $\langle\mathrm{U}_2\rangle_3$. Then each party P_i adds these three shared matrices locally to compute $\langle\mathrm{U}\rangle_i = \langle\mathrm{U}_0\rangle_i + \langle\mathrm{U}_1\rangle_i + \langle\mathrm{U}_2\rangle_i$. Additionally, P_0 computes $\langle\mathrm{U}\rangle_3 = \langle\mathrm{U}_0\rangle_3 + \langle\mathrm{U}_1\rangle_3 + \langle\mathrm{U}_2\rangle_3$. For the generation of $\langle\mathrm{H}\rangle$, we generalize the secure computation method proposed by Zhu and Takagi [40] to shared matrices. Firstly, P_0 and P_1 interactively compute $[\mathrm{U}_0 \times \mathrm{V}_1 + \mathrm{U}_1 \times \mathrm{V}_0]$, P_0 and P_2 interactively compute $[\mathrm{U}_0 \times \mathrm{V}_2 + \mathrm{U}_2 \times \mathrm{V}_0]$, P_1 and P_2 interactively compute $[\mathrm{U}_1 \times \mathrm{V}_2 + \mathrm{U}_2 \times \mathrm{V}_1]$. Then each party P_i locally computes $\mathrm{H}_i = \mathrm{U}_i \times \mathrm{V}_i + [\mathrm{U}_i \times \mathrm{V}_{i+1} + \mathrm{U}_{i+1} \times \mathrm{V}_i]_i + [\mathrm{U}_i \times \mathrm{V}_{i-1} + \mathrm{U}_{i-1} \times \mathrm{V}_i]_i$. Furthermore, each party P_i shares H_i using the matrix sharing protocol. Finally, each party P_i locally computes $\langle\mathrm{H}\rangle_i = \langle\mathrm{H}_0\rangle_i + \langle\mathrm{H}_1\rangle_i + \langle\mathrm{H}_2\rangle_i$. P_0 additionally computes the alternate shared matrix $\langle\mathrm{H}\rangle_3 = \langle\mathrm{H}_0\rangle_3 + \langle\mathrm{H}_1\rangle_3 + \langle\mathrm{H}_2\rangle_3$.

Protocol 6 $\prod_{\mathrm{trunc}}(\mathcal{P}, \langle z\rangle)$

Preprocessing: Parties pre-shared random values $\langle r\rangle$ and $\langle r'\rangle = \langle r/2^{\ell_f}\rangle$
Input: $\langle z\rangle$
Output: The result after truncation $\langle z'\rangle$, where $z' = z/2^{\ell_f}$

```
1: $P_i$ locally computes $\langle z - r\rangle_i = \langle z\rangle_i - \langle r\rangle_i$. $P_0$ additionally computes $\langle z - r\rangle_3 = \langle z\rangle_3 - \langle r\rangle_3$;
2: $P_1$ and $P_2$ send $\langle z - r\rangle_1$ and $\langle z - r\rangle_2$ to $P_0$ respectively.
3: $P_0$ locally computes $\langle z'\rangle_0 = (z - r)/(2^{\ell_f} \cdot c_0) + \langle r'\rangle_0$ and *assistant parties* $P_j$ for $j \in \{1, 2\}$ holds $\langle z\rangle_j = \langle r'\rangle_j$. $P_0$ additionally holds $\langle z'\rangle_3 = \langle r'\rangle_3$.
```

Truncation: After multiplying two fixed-point numbers with ℓ_f bits in the fractional part, the fractional part of the computation result is extended to $2\ell_f$ bits. In order to return the result of the multiplication back to the same format as that of the inputs, parties interactively execute the truncation on the result of the multiplication.

Protocol 6 shows the truncation protocol $\prod_{\mathrm{trunc}}(\mathcal{P}, \langle z \rangle)$ proposed in pMPL. At first, we observe that:

$$
\begin{aligned}
z' = \frac{z}{2^{\ell_f}} &= \frac{c_0 \cdot \langle z \rangle_0 + c_1 \cdot \langle z \rangle_1 + c_2 \cdot \langle z \rangle_2}{2^{\ell_f}} \\
&= \frac{\begin{array}{l} c_0 \cdot (\langle z \rangle_0 - \langle r \rangle_0 + \langle r \rangle_0) + c_1 \cdot (\langle z \rangle_1 - \langle r \rangle_1 + \langle r \rangle_1) + \\ c_2 \cdot (\langle z \rangle_2 - \langle r \rangle_2 + \langle r \rangle_2) \end{array}}{2^{\ell_f}} \\
&= \frac{(z - r) + c_0 \cdot \langle r \rangle_0 + c_1 \cdot \langle r \rangle_1 + c_2 \cdot \langle r \rangle_2}{2^{\ell_f}} \\
&= \frac{z - r}{2^{\ell_f}} + c_0 \cdot \frac{\langle r \rangle_0}{2^{\ell_f}} + c_1 \cdot \frac{\langle r \rangle_1}{2^{\ell_f}} + c_2 \cdot \frac{\langle r \rangle_2}{2^{\ell_f}} \\
&= c_0 \cdot \frac{(z - r)/c_0 + \langle r \rangle_0}{2^{\ell_f}} + c_1 \cdot \frac{\langle r \rangle_1}{2^{\ell_f}} + c_2 \cdot \frac{\langle r \rangle_2}{2^{\ell_f}}
\end{aligned}
\tag{13}
$$

We assume that parties have held the shares $\langle r \rangle$ and $\langle r' \rangle = \langle r/2^{\ell_f} \rangle$. To compute the shares of $z' = z/2^{\ell_f} = (x \cdot y)/2^{\ell_f}$, P_1 and P_2 sends $\langle z - r \rangle_1$ and $\langle z - r \rangle_2$ to P_0 respectively. Then P_0 locally computes $z - r = c_0 \cdot \langle z - r \rangle_0 + c_1 \cdot \langle z - r \rangle_1 + c_2 \cdot \langle z - r \rangle_2$, $(z - r)/(2^{\ell_f} \cdot c_0) + \langle r' \rangle_0$, and P_1, P_2 hold $\langle r' \rangle_1$,$\langle r' \rangle_2$, respectively. Additionally, P_0 holds $\langle r' \rangle_3$. Finally, the shares $\langle z \rangle$ are truncated.

For truncation pairs, we use some edabits [12] to generate them. The edabits are used in the share conversation between $[\cdot]$ and $[\cdot]^2$. An edabit consists of a value r in $\mathbb{Z}_{2^\ell}$, together with a set of ℓ random bits $(r_0, \ldots, r_{\ell-1})$ shared in the boolean world, where $r = \sum_{i=0}^{\ell-1} 2^i \cdot r_i$. $\prod_{\mathrm{trunpair}}(\mathcal{P})$ (Protocol 7) shows how to generate truncation pairs. Firstly, parties generate edabits $([r], [r_0]^2, [r_1]^2, \ldots, [r_{\ell-1}]^2)$ and $([r'], [r'_0]^2, [r'_1]^2, \ldots, [r'_{\ell-\ell_f-1}]^2)$, where $r' = r/2^{\ell_f}$. After that, each party holds $[\cdot]$-sharing of r. Then they interactively execute $\prod_{\mathrm{a2v}}(\mathcal{P}, [r])$ and $\prod_{\mathrm{a2v}}(\mathcal{P}, [r'])$ (Protocol 5) to get $\langle r \rangle$ and $\langle r' \rangle$.

Protocol 7 $\prod_{\mathrm{trunpair}}(\mathcal{P})$

```
Input: $\emptyset$
Output: The truncation pairs $(\langle r \rangle, \langle r' \rangle)$, where $r' = r/2^{\ell_f}$
1: Parties generate edabits $[r], [r_0]^2, [r_1]^2, \ldots, [r_{\ell-1}]^2$ and $[r'], [r'_0]^2, [r'_1]^2, \ldots, [r'_{\ell-\ell_f-1}]^2$.
2: Parties interactively execute protocol $\prod_{\mathrm{a2v}}(\mathcal{P}, [r])$ and $\prod_{\mathrm{a2v}}(\mathcal{P}, [r'])$ (Protocol 5).
```

Activation Functions: We consider two widely used non-linear activation functions in machine learning, i.e. ReLU and Sigmoid. Besides, we describe the approximations and computations of these activation functions in pMPL as follows.

- *ReLU:* ReLU function, which is defined as $\texttt{ReLU(x)} = max(x, 0)$, can be viewed as $\texttt{ReLU(x)} = (1 \oplus b) \cdot x$. The bit b denotes the MSB of x, where $b = 1$ if $x < 0$ and 0 otherwise. $\prod_{\mathrm{relu}}(\mathcal{P}, \langle x \rangle)$ (Protocol 8) enables parties to compute the shares of ReLU function outputs, $\langle \texttt{ReLU(x)} \rangle$. Firstly, parties interactively execute $\prod_{\mathrm{v2a}}(\mathcal{P}, \langle x \rangle)$ to convert $\langle x \rangle$ to $[x]$. Then they interactively execute $\prod_{\mathrm{msbext}}(\mathcal{P}, [x])$ on $[x]$ to obtain the share of MSB of x, namely $[b]^2$. Furthermore, each party P_i locally computes $[1 \oplus b]^2$. Next, parties interactively execute $\prod_{\mathrm{b2a}}(\mathcal{P}, [1 \oplus b]^2)$ to convert $[1 \oplus b]^2$ to $[1 \oplus b]$. After that, parties interactively execute $\prod_{\mathrm{a2v}}(\mathcal{P}, [1 \oplus b])$ (Protocol 5) to convert $[1 \oplus b]$ to $\langle 1 \oplus b \rangle$.

Protocol 8 $\prod_{\mathrm{relu}}(\mathcal{P}, \langle x \rangle)$

```
Input: $\langle x \rangle$
Output: $\langle \texttt{ReLU(x)} \rangle$, where $\texttt{ReLU(x)} = 0$ if $x < 0$ and $x$ otherwise
1: Parties locally execute $\prod_{\mathrm{v2a}}(\mathcal{P}, \langle x \rangle)$ to obtain $[x]$.
2: Parties interactively execute $\prod_{\mathrm{msbext}}(\mathcal{P}, [x])$ to obtain $[b]^2$.
3: $P_i$ computes $[1 \oplus b]^2$ locally.
4: Parties interactively execute $\prod_{\mathrm{b2a}}(\mathcal{P}, [1 \oplus b]^2)$ to obtain $[1 \oplus b]$.
5: Parties interactively execute $\prod_{\mathrm{a2v}}(\mathcal{P}, [1 \oplus b])$ (Protocol 5) to obtain $\langle 1 \oplus b \rangle$.
6: Parties interactively execute $\prod_{\mathrm{mul}}(\mathcal{P}, \langle 1 \oplus b \rangle, \langle x \rangle)$ (Protocol 3) to compute $\langle \texttt{ReLU(x)} \rangle$
```

At last, parties interactively execute $\prod_{\mathrm{mul}}(\mathcal{P}, \langle 1 \oplus b \rangle, \langle x \rangle)$ (Protocol 3) to compute $\langle \texttt{ReLU(x)} \rangle$, such that $\texttt{ReLU(x)} = 0$ if $x < 0$, and $\texttt{ReLU(x)} = x$ otherwise.

- *Sigmoid:* Sigmoid function is defied as $\texttt{Sigmoid}(x) = 1/(1 + e^{-x})$. In this paper, we use an MPC-friendly version [26] of the Sigmoid function, which is defined as:

$$
\texttt{Sigmoid}(x) = \begin{cases} 0, & x \le -\frac{1}{2} \\ x + \frac{1}{2}, & -\frac{1}{2} < x < \frac{1}{2} \\ 1, & x \ge \frac{1}{2} \end{cases}
\tag{14}
$$

This function can be viewed as $\texttt{Sigmoid}(x) = (1 \oplus b_1) \cdot b_2 \cdot (x + 1/2) + (1 \oplus b_2)$, where $b_1 = 1$ if $x < -1/2$ and $b_2 = 1$ if $x < 1/2$. $\prod_{\mathrm{sig}}(\mathcal{P}, \langle x \rangle)$ is similar to $\prod_{\mathrm{relu}}(\mathcal{P}, \langle x \rangle)$. We thus do not describe it in detail.

4.5 Robustness Design (2PC)

In pMPL, we ensure the robustness through the design of the alternate shares. If P_2 drops out, the alternate shares will replace the shares held by P_2. Therefore, even if one *assistant party* (P_2) drops out, the remaining two parties (P_0 and P_1) can continue training. Here, we describe the protocols for the scenario of one of two *assistant parties* (P_2) drops out, i.e. 2PC protocols.

Secure Addition and Secure Multiplication: To get the result of secure addition $\langle x + y \rangle$, if P_2 drops out, P_0 locally computes $\langle z \rangle_0 = \langle x \rangle_0 + \langle y \rangle_0$, $\langle z \rangle_3 = \langle x \rangle_3 + \langle y \rangle_3$, and P_1 locally computes $\langle z \rangle_1 = \langle x \rangle_1 + \langle y \rangle_1$.

Protocol 9 $\prod_{\mathrm{mul2}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$

```
Preprocessing: Parties pre-shared vector multiplication triplet
$\langle u \rangle, \langle v \rangle, \langle h \rangle$ using $\prod_{\mathrm{vmtgen}}(\mathcal{P})$ (Protocol 4)
Input: $\langle x \rangle$ and $\langle y \rangle$.
Output: $\langle x \cdot y \rangle$.
1: $P_j$ for $j \in \{0, 1\}$ locally computes $\langle e \rangle_j = \langle x \rangle_j + \langle u \rangle_j$ and $\langle f \rangle_j = \langle y \rangle_j + \langle v \rangle_j$. Besides, $P_0$ computes $\langle e \rangle_3 = \langle x \rangle_3 + \langle u \rangle_3$ and $\langle f \rangle_3 = \langle y \rangle_3 + \langle v \rangle_3$.
2: Parties interactively execute $\prod_{\mathrm{rec}}(\mathcal{P}, \langle e \rangle)$ (Protocol 2) and $\prod_{\mathrm{rec}}(\mathcal{P}, \langle f \rangle)$ (Protocol 2).
3: $P_j$ for $j \in \{0, 1\}$ locally computes $\langle z \rangle_j = \langle x \rangle_j \cdot f - \langle v \rangle_j \cdot e + \langle h \rangle_j$. Besides, $P_0$ computes $\langle z \rangle_3 = \langle x \rangle_3 \cdot f - \langle v \rangle_3 \cdot e + \langle h \rangle_3$.
```

Protocol 9 shows 2PC secure multiplication protocol $\prod_{\mathrm{mul2}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$. Firstly, P_0 locally computes $\langle e \rangle_0 = \langle x \rangle_0 + \langle u \rangle_0$,

$\langle e\rangle_3 = \langle x\rangle_3 + \langle u\rangle_3$ and $\langle f\rangle_0 = \langle y\rangle_0 + \langle v\rangle_0$, $\langle f\rangle_3 = \langle y\rangle_3 + \langle v\rangle_3$. P_1 also locally computes $\langle e\rangle_1 = \langle x\rangle_1 + \langle u\rangle_1$ and $\langle f\rangle_1 = \langle y\rangle_1 + \langle v\rangle_1$. Then P_0 and P_1 interactively execute $\prod_{\mathrm{rec}}(\mathcal{P}, \langle e\rangle)$ (Protocol 2) and $\prod_{\mathrm{rec}}(\mathcal{P}, \langle f\rangle)$ (Protocol 2) to obtain e and f respectively. Finally, P_0 computes $\langle z\rangle_0 = \langle x\rangle_0 \cdot f - \langle v\rangle_0 \cdot e + \langle h\rangle_0$, $\langle z\rangle_3 = \langle x\rangle_3 \cdot f - \langle v\rangle_3 \cdot e + \langle h\rangle_3$, and P_1 computes $\langle z\rangle_1 = \langle x\rangle_1 \cdot f - \langle v\rangle_1 \cdot e + \langle h\rangle_1$.

Sharing Conversion: If P_2 drops out, it is trivial to see that the conversions between $\langle\cdot\rangle$-sharing and $[\cdot]$-sharing and conversions between $[\cdot]$-sharing and $\langle\cdot\rangle$-sharing can be done by P_0 and P_1 locally.

- *Converting $\langle\cdot\rangle$-sharing to $[\cdot]$-sharing:* P_0 locally computes $[x]_0 = c'_0 \cdot \langle x\rangle_0$ and $[x]_3 = c'_3 \cdot \langle x\rangle_3$. Besides, P_1 locally computes $[x]_1 = c'_1 \cdot \langle x\rangle_1$, such that $x = c'_0 \cdot \langle x\rangle_0 + c'_1 \cdot \langle x\rangle_1 + c'_3 \cdot \langle x\rangle_3 = [x]_0 + [x]_1 + [x]_3$. Therefore, P_0 and P_1 convert their $\langle\cdot\rangle$-shares to $[\cdot]$-shares.
- *Converting $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing:* P_0 locally computes $\langle x\rangle_0 = [x]_0/c'_0$ and $\langle x\rangle_3 = [x]_3/c'_3$. Besides, P_1 locally computes $\langle x\rangle_1 = [x]_1/c'_1$.

Protocol 10 $\prod_{\mathrm{trunc2}}(\mathcal{P}, \langle z\rangle)$

Preprocessing: Parties pre-shared random values $\langle r\rangle$ and $\langle r'\rangle = \langle r/2^{\ell_f}\rangle$
Input: $\langle z\rangle$
Output: The result after truncation $\langle z'\rangle$, where $z' = z/2^{\ell_f}$

```
1: $P_j$ for $j \in \{0,1\}$ locally computes $\langle z-r\rangle_j = \langle z\rangle_j - \langle r\rangle_j$. $P_0$ also computes $\langle z-r\rangle_3 = \langle z\rangle_3 - \langle r\rangle_3$;
2: $P_1$ sends $\langle z-r\rangle_1$ to $P_0$.
3: $P_0$ locally computes $\langle z'\rangle_0 = (z-r)/(2^{\ell_f} \cdot c'_0) + \langle r'\rangle_0$ and holds $\langle z'\rangle_3 = \langle r'\rangle_3$. $P_1$ holds $\langle z'\rangle_1 = \langle r'\rangle_1$.
```

Truncation: If P_2 drops out, Equation (13) can be rewritten as:

$$z' = c'_0 \cdot \frac{(z-r)/c'_0 + \langle r\rangle_0}{2^{\ell_f}} + c'_1 \cdot \frac{\langle r\rangle_1}{2^{\ell_f}} + c'_3 \cdot \frac{\langle r\rangle_3}{2^{\ell_f}} \qquad (15)$$

Protocol 10 shows the 2PC secure truncation protocol $\prod_{\mathrm{trunc2}}(\mathcal{P}, \langle z\rangle)$. Firstly, P_1 sends $\langle z-r\rangle_1$ to P_0. Then P_0 locally computes $z - r = c'_0 \cdot \langle z-r\rangle_0 + c'_1 \cdot \langle z-r\rangle_1 + c'_3 \cdot \langle z-r\rangle_3$ and $(z-r)/(2^{\ell_f} \cdot c'_0) + \langle r'\rangle_0$. Besides, P_0 also holds $\langle r'\rangle_3$ and P_1 holds $\langle r'\rangle_1$. Note that matrix addition and matrix multiplication protocols for 2PC generalize secure addition and secure multiplication protocols for 2PC. These protocols are similar to the ones for 3PC. In addition, MSB extraction and Bit2A protocols for 2PC are the same as the ones for 3PC.

4.6 Complexity Analysis

We measure the cost of each building block from two aspects: online communication rounds and online communication size in both 3PC (no party drops out) and 2PC (P_2 drops out) settings. Table 2 shows the comparison of the communication rounds and communication size among pMPL, SecureML and TF-Encrypted.

5 EVALUATION

In this section, we present the implementation of linear regression, logistic regression and neural networks in pMPL. Meanwhile, we conduct experiments to evaluate the performance of pMPL by the comparison with other MPL frameworks.

Table 2: Communication rounds and total communication size (bit) cost of building blocks in pMPL, SecureML and TF-Encrypted. Here, ℓ denotes the number of bits of a value. $n \times d$, $d \times m$ are the sizes for the left and right inputs of matrix-based computations. ReLU and Sigmoid are executed on a single value. λ is the security parameter of oblivious transfer used in SecureML. Rounds stands for online communication rounds and Comm. stands for online communication size.

Building block	Framework	3PC		2PC	
		Rounds	Comm.	Rounds	Comm.
Matrix addition	pMPL	0	0	0	0
	SecureML	\	\	0	0
	TF-Encrypted	0	0	\	\
Matrix multiplication	pMPL	1	$6\ell(nd+dm)$	1	$3\ell(nd+dm)$
	SecureML	\	\	1	$2\ell(nd+dm)$
	TF-Encrypted	1	$3\ell nm$	\	\
Matrix truncation	pMPL	1	$2\ell nm$	1	ℓnm
	SecureML	\	\	0	0
	TF-Encrypted	1	$2\ell nm$	\	\
Multiplication with truncation	pMPL	2	$6\ell(nd+dm)+2\ell nm$	2	$\ell nm+3\ell(nd+dm)$
	SecureML	\	\	1	$2\ell(nd+dm)$
	TF-Encrypted	1	$4\ell nm$	\	\
ReLU	pMPL	$\log\ell+5$	$18\ell+4\ell\log\ell$	$\log\ell+4$	$8\ell+2\ell\log\ell$
	SecureML	\	\	2	$4\lambda(\ell-1)+2(\ell+\lambda)$
	TF-Encrypted	$\log\ell+1$	$3\ell+3\ell\log\ell$	\	\
Sigmoid	pMPL	$\log\ell+6$	$38\ell+8\ell\log\ell$	$\log\ell+5$	$18\ell+4\ell\log\ell$
	SecureML	\	\	4	$4\lambda(2\ell-1)+6\ell$
	TF-Encrypted	$\log\ell+3$	$9\ell+3\ell\log\ell$	\	\

5.1 Experiment Settings and Datasets

Experiment Settings: We conduct 3PC experiments on three Linux servers equipped with 20-core 2.4 Ghz Intel Xeon CPUs and 128GB of RAM, and 2PC experiments on two Linux servers equipped same as above. The experiments are performed on two network environments: one is the LAN setting with a bandwidth of 1Gbps and sub-millisecond RTT (round-trip time) latency, the other one is the WAN setting with 40Mbps bandwidth and 40ms RTT latency. Note that we run TF-Encrypted (with ABY3 as the back-end framework) under the above environment. While the experimental results of SecureML are from the study [26] and [24] since the code of SecureML is not public. We implement pMPL in C++ over the ring $\mathbb{Z}_{2^\ell}$. Here, we set $\ell = 64$, and the least 20 significant bits ℓ_f represent the fractional part, which is the same as the setting of SecureML and TF-Encrypted. Additionally, we set *public matrix* $\Phi(\mathcal{P})$ as follows:

$$\Phi(\mathcal{P}) = \begin{bmatrix} \Phi(0) \\ \Phi(1) \\ \Phi(2) \\ \Phi(3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 2^\ell - 1 \\ 2 & 2 & 2^\ell - 3 \\ 3 & 3 & 2^\ell - 4 \end{bmatrix}$$

Therefore, according to Equation (6), we can compute $c_0 = 1, c_1 = 2^\ell - 2, c_2 = 1, c'_0 = 1, c'_1 = 2^\ell - 3, c'_3 = 1, c''_0 = 1, c''_2 = 3, c''_3 = 2^\ell - 2$.

Datasets: To evaluate the performance of pMPL, we use the MNIST dataset[21]. It contains image samples of handwritten digits from "0" to "9", each with 784 features representing 28 × 28 pixels. Besides, the greyscale of each pixel is between 0~255. Its training set contains 60,000 samples, and the testing set contains 10,000 samples. For linear regression and logistic regression, we consider binary classification, where the digits "0" as a class, and the digits "1 ∼ 9" as another one. For BP neural network, we consider a ten-class classification task. Additionally, we benchmark more complex datasets, including Fashion-MNIST [35] and SVHN [27], in Appendix C.

5.2 Offline Phase

We evaluate the performance of generating the vector multiplication triplets under the LAN setting in the offline phase. We follow the same setting as SecureML, where the batch size $B = 128$, epoch $E = 2$, the number of samples $n \in \{100, 1,000, 10,000\}$ and the dimension $D \in \{100, 500, 1,000\}$. The number of iterations is $n*E/B$. As is shown in Table 3, pMPL is faster than both SecureML based on HE protocol and SecureML based on OT protocol. Especially when the dimension $D = 1,000$ and number of samples $n = 10,000$, pMPL is around 119× faster than SecureML based on HE protocol and around 6× faster than SecureML based on OT protocol.

Table 3: Performance of the offline phase (*seconds*). $*$ means estimated via extrapolation.

Number of samples n	Protocol	Dimension (D)		
		100	500	1,000
1,000	pMPL	0.34	0.78	1.33
	SecureML (HE-based)	23.9	83.9	158.4
	SecureML(OT-based)	0.86	3.8	7.9
10,000	pMPL	3.73	7.89	13.21
	SecureML (HE-based)	248.4	869.1	1600.9
	SecureML(OT-based)	7.9	39.2	80.0
100,000	pMPL	38.05	78.70	140.28
	SecureML (HE-based)	2437.1	8721.5	16000^*
	SecureML(OT-based)	88.0	377.9	794.0

5.3 Secure Training in Online Phase

As is mentioned in Section 2.3, the training of the evaluated machine learning models consists of two phases: (1) the forward propagation phase is to compute the output; (2) the backward propagation phase is to update coefficient parameters according to the error between the output computed in the forward propagation and the actual label. One iteration in the training phase contains one forward propagation and a backward propagation.

To compare pMPL with SecureML and TF-Encrypted, we select $D \in \{10, 100, 1,000\}$ and $B \in \{128, 256, 512, 1,024\}$. In addition, we consider two scenarios for experiments, i.e. 3PC with no *assistant party* drops out, and 2PC with P_2 drops out.

Linear Regression: We use mini-batch stochastic gradient descent (SGD for short) to train a linear regression model. The update function in Equation (4) can be expressed as:

$$\vec{w} := \vec{w} - \frac{\alpha}{B}\mathrm{X}_i^T \times (\mathrm{X}_i \times \vec{w} - \mathrm{Y_i})$$

where $\mathrm{X_i}$ is a subset of batch size B. Besides, $(\mathrm{X}_i, \mathrm{Y}_i)$ are randomly selected from the whole dataset in the i-th iteration.

As is shown in Table 4, the experimental results show that:

(1) In the LAN setting, pMPL for 3PC is around 2.7× ∼ 16.1× faster and pMPL for 2PC is around 3.8× ∼ 18.6× faster than TF-Encrypted. We analyze that this is due to Tensorflow, which is the basis of TF-Encrypted, bringing some extra overhead, e.g. operator schedulings. As the training process of linear regression is relatively simple, when we train linear regression with TF-Encrypted, the extra overhead brought by Tensorflow becomes the main performance bottleneck. Besides, SecureML is faster than pMPL. The performance differences between pMPL and SecureML are led by two reasons. First of all, the experiment environments are different. As the source code of SecureML is not available, the experimental results of SecureML, which are obtained in the different environment with pMPL, are from the study [24]. More specifically, we perform our experiment on 2.4 Ghz Intel Xeon CPUs and 128GB of RAM, while the study [24] performs on 2.7 Ghz Intel Xeon CPUs and 256GB of RAM, which leads to the local computing of SecureML being faster than pMPL. Meanwhile, our bandwidth is 1Gbps, while the bandwidth of the study [24] is 10 Gbps. Second, the underlying techniques are different. The online communication overhead of building blocks in pMPL is more than those in SecureML (as shown in Table 2). For instance, the truncation operation in pMPL needs one round while SecureML performs the truncation operation locally without communication.

Table 4: Online throughput of linear regression compared to SecureML and TF-Encrypted (iterations/second).

Setting	Dimension (D)	Protocol	Batch Size (B)			
			128	256	512	1,024
LAN	10	pMPL (3PC)	4545.45	3846.15	2631.58	1666.67
		pMPL (2PC)	5263.16	4166.67	2777.78	1694.92
		SecureML	7,889	7,206	4,350	4,263
		TF-Encrypted	282.36	248.47	195.18	139.51
	100	pMPL (3PC)	1333.33	740.74	387.60	166.67
		pMPL (2PC)	1428.57	813.01	436.68	202.02
		SecureML	2,612	755	325	281
		TF-Encrypted	141.17	90.95	55.36	30.06
	1,000	pMPL (3PC)	89.05	39.53	17.74	8.87
		pMPL (2PC)	137.36	58.82	26.39	12.43
		SecureML	131	96	45	27
		TF-Encrypted	24.53	12.74	6.55	3.30
WAN	10	pMPL (3PC)	4.93	4.89	4.84	4.73
		pMPL (2PC)	4.94	4.921	4.88	4.80
		SecureML	12.40	12.40	12.40	12.40
		TF-Encrypted	11.58	11.53	11.42	11.15
	100	pMPL (3PC)	4.66	4.47	4.10	3.55
		pMPL (2PC)	4.75	4.67	4.30	4.03
		SecureML	12.30	12.20	11.80	11.80
		TF-Encrypted	11.13	10.63	9.74	8.32
	1,000	pMPL (3PC)	3.29	2.47	1.51	0.84
		pMPL (2PC)	3.83	3.14	2.11	1.32
		SecureML	11.00	9.80	9.20	7.30
		TF-Encrypted	7.85	5.76	3.80	2.22

(2) In the WAN setting, SecureML and TF-Encrypted are faster than pMPL. This is because to provide more security guarantees (i.e., defending the collusion of two assistant parties) and ensure robustness, pMPL requires more communication overhead than SecureML and TF-Encrypted (as shown in Table 2). Therefore, the performance of pMPL is promising.

(3) In the both LAN setting and WAN setting, pMPL for 2PC is faster than 3PC. This is because the communication overhead of 2PC is smaller.

Besides, the trained model can reach an accuracy of 97% on the test dataset.

Logistic Regression: Similar to linear regression, the update function using mini-batch SGD method in logistic regression can be expressed as:

$$\vec{w} := \vec{w} - \frac{\alpha}{B}\mathrm{X}_i^T \times (\mathtt{Sigmoid}(\mathrm{X}_i \times \vec{w}) - \mathrm{Y_i})$$

As is shown in Table 5, the experimental results show that:

(1) In the LAN setting, pMPL is faster than both SecureML and TF-Encrypted. The reason for these performance differences between pMPL and SecureML is SecureML implements Sigmoid utilizing the garbled circuit and oblivious transfer. It requires fewer communication rounds but much bigger communication size than those in pMPL (as shown in Table 2). Besides, the reasons for these

Table 5: Online throughput of logistic regression compared to SecureML and TF-Encrypted (iterations/second).

Setting	Dimension (D)	Protocol	Batch Size (B)			
			128	256	512	1,024
LAN	10	pMPL (3PC)	579.45	537.47	444.45	330.40
		pMPL (2PC)	598.75	542.68	455.19	332.68
		SecureML	188	101	41	25
		TF-Encrypted	119.88	110.78	97.16	74.07
	100	pMPL (3PC)	425.88	332.86	222.89	121.92
		pMPL (2PC)	435.41	353.55	235.93	128.25
		SecureML	183	93	46	24
		TF-Encrypted	87.34	63.06	41.25	25.12
	1,000	pMPL (3PC)	100.66	49.53	22.85	11.18
		pMPL (2PC)	105.82	51.62	23.37	11.40
		SecureML	105	51	24	13.50
		TF-Encrypted	22.10	12.07	6.42	3.28
WAN	10	pMPL (3PC)	0.65	0.64	0.63	0.62
		pMPL (2PC)	0.65	0.65	0.64	0.63
		SecureML	3.10	2.28	1.58	0.99
		TF-Encrypted	4.92	4.91	4.90	4.81
	100	pMPL (3PC)	0.63	0.62	0.60	0.56
		pMPL (2PC)	0.64	0.63	0.62	0.60
		SecureML	3.08	2.25	1.57	0.99
		TF-Encrypted	4.83	4.69	4.59	4.21
	1,000	pMPL (3PC)	0.56	0.52	0.42	0.32
		pMPL (2PC)	0.60	0.57	0.51	0.42
		SecureML	3.01	2.15	1.47	0.93
		TF-Encrypted	4.05	3.47	2.65	1.76

performance differences between pMPL and TF-Encrypted are the same as those for linear regression.

(2) In the WAN setting, SecureML and TF-Encrypted are faster than pMPL. This is because the communication rounds are important performance bottlenecks in the WAN setting. Meanwhile, pMPL requires more communication rounds than SecureML and TF-Encrypted (as shown in Table 2) to provide more security guarantees (i.e., defending the collusion of two assist parties) and ensure robustness. Therefore, the performance of pMPL is promising.

(3) pMPL for 2PC is faster than 3PC. This is also because the communication overhead of 2PC is smaller.

Besides, the trained model can reach an accuracy of 99% on the test dataset.

BP Neural Networks: For BP neural networks, we follow the steps similar to those of SecureML and TF-Encrypted. In pMPL, we consider a classical BP neural network consisting of four layers, including one input layer, two hidden layers, and one output layer. Besides, we use ReLU as the activation function. As is shown in Table 6, the experimental results show that:

(1) TF-Encrypted is faster than pMPL. When we train BP neural networks, which are more complex than linear regression and logistic regression, the overhead of model training becomes the performance bottleneck in TF-Encrypted rather than the extra overhead brought by Tensorflow. Meanwhile, pMPL requires more communication overhead (as shown in Table 2) than TF-Encrypted to provide more security guarantees (i.e., defending the collusion of two assist parties) and ensure robustness, two requirements from novel practical scenarios. The performance of pMPL is still promising.

(2) pMPL for 2PC is faster than 3PC. This is also because the communication overhead of 2PC is smaller.

After training the neural network on MNIST dataset with batch size $B = 128$, dimension $D = 784$, pMPL can reach the accuracy of 96% on the test dataset.

Table 6: Online throughput of BP neural networks compared to TF-Encrypted (iterations/second).

Setting	Dimension (D)	Protocol	Batch Size (B)			
			128	256	512	1,024
LAN	10	pMPL (3PC)	16.49	8.43	4.08	1.86
		pMPL (2PC)	17.61	8.62	4.14	1.91
		TF-Encrypted	29.56	18.95	11.38	6.13
	100	pMPL (3PC)	15.79	7.88	3.84	1.77
		pMPL (2PC)	16.23	8.17	3.95	1.81
		TF-Encrypted	25.39	15.78	8.63	5.02
	1,000	pMPL (3PC)	8.93	5.25	2.65	1.29
		pMPL (2PC)	9.19	5.33	2.66	1.31
		TF-Encrypted	12.38	6.89	3.54	1.80
WAN	10	pMPL (3PC)	0.15	0.12	0.10	0.07
		pMPL (2PC)	0.16	0.14	0.12	0.09
		TF-Encrypted	0.93	0.65	0.40	0.22
	100	pMPL (3PC)	0.15	0.12	0.10	0.07
		pMPL (2PC)	0.16	0.14	0.12	0.09
		TF-Encrypted	0.92	0.64	0.39	0.21
	1,000	pMPL (3PC)	0.14	0.12	0.09	0.06
		pMPL (2PC)	0.15	0.13	0.11	0.08
		TF-Encrypted	0.80	0.55	0.33	0.18

6 DISCUSSION

6.1 pMPL with More Assistant Parties

Our proposed pMPL can be extended to support more *assistant parties* by setting *pubic matrix* $\Phi(\mathcal{P})$. In order to support more *assistant parties*, we can increase the number of columns of the *public matrix* $\Phi(\mathcal{P})$, i.e. expand the dimension of each *public vector* $\Phi(i)$. For instance, for a set of parties $\mathcal{P} = \{P_0, P_1, P_2, P_3, P_4\}$ and an access structure $\Gamma = \{B_0, B_1, B_2, B_3, B_4\} = \{\{P_0, P_1, P_2, P_3, P_4\}, \{P_0, P_2, P_3, P_4\}, \{P_0, P_1, P_3, P_4\}, \{P_0, P_1, P_2, P_4\}, \{P_0, P_1, P_2, P_3\}\}$, where P_0 is the *privileged party* and P_1, P_2, P_3, P_4 are *assistant parties*. The secret cannot be revealed without the participation of the *privileged party* P_0, even when *assistant parties* collude and one of *assistant parties* drops out during training.

To securely perform the training in the above application scenario, the *public matrix* $\Phi(\mathcal{P})$ with the size of 6×5 should satisfy the following four restrictions:

- $(1, 0, 0, 0, 0)$ can be written as a linear combination of *public vectors* in the set $\{\Phi(0), \Phi(1), \Phi(2), \Phi(3), \Phi(4)\}$, where all *public vectors* are linear independent.
- The alternate *public vector* $\Phi(5)$ held by the *privileged party* P_0 can be represented linearly by *public vectors* $\Phi(1), \Phi(2), \Phi(3)$ and $\Phi(4)$. That is, $\Phi(5) = \sum_{j=1}^{4} a_j * \Phi(j)$, where $j \in \{1, 2, 3, 4\}$ and $a_j \neq 0$. Therefore, $(1, 0, 0, 0, 0)$ can also be a linear combination of the *public vectors* in sets $\{\Phi(0), \Phi(2), \Phi(3), \Phi(4), \Phi(5)\}$, $\{\Phi(0), \Phi(1), \Phi(3), \Phi(4), \Phi(5)\}$, $\{\Phi(0), \Phi(1), \Phi(2), \Phi(4), \Phi(5)\}$, $\{\Phi(0), \Phi(1), \Phi(2), \Phi(3), \Phi(5)\}$, respectively.
- To guarantee that only the set of parties in the access structure can collaboratively reveal the secret value, $(1, 0, 0, 0, 0)$ cannot be represented as a linear combination of *public vectors* in the sets $\{\Phi(1), \Phi(2), \Phi(3), \Phi(4), \Phi(5)\}$, $\{\Phi(0), \Phi(5)\}$ and their subsets.
- The values of *public matrix* $\Phi(\mathcal{P})$ and reconstruction coefficients should be elements of the ring $\mathbb{Z}_{2^\ell}$.

For example, a *public matrix* $\Phi(\mathcal{P})$ that satisfies the above restrictions is:

$$\Phi(\mathcal{P}) = \begin{bmatrix} \Phi(0) \\ \Phi(1) \\ \Phi(2) \\ \Phi(3) \\ \Phi(4) \\ \Phi(5) \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 \\ 2^{\ell}-1 & 1 & 0 & 1 & 3 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 2 & 1 & 4 & 9 \end{bmatrix}.$$

Note that we can hereby tolerate more *assistant parties* (≤ 3) dropping out during the training by setting more alternate vectors for the *privileged party* P_0. Furthermore, when more *assistant parties* are involved, the protocols proposed in Section 4 can be directly used with simple extensions.

6.2 Comparison with the MPL Frameworks based on Additive Secret Sharing

In the MPL frameworks [26, 34], such as SecureML [26], SecureNN [34], based on additive secret sharing [3], the final model can be revealed only when all parties corporate. Thus, these additive secret sharing based MPL frameworks can meet the first requirement mentioned in Section 1 by setting a sole party to hold all trained shares. However, these additive secret sharing based frameworks cannot meet the second requirement. In these MPL frameworks, once one party drops out, the training will be aborted and must be restarted. Especially, when one party in additive secret sharing based MPL frameworks, e.g. SecureML, intentionally quit the training, the training process cannot be restarted.

In our proposed pMPL, which is based on vector space secret sharing, the chances of handling the result between the *privileged party* and *assistant parties* are different. Because every authorized set contains the *privileged party* P_0, without the participation of P_0, *assistant parties* cannot reveal the secret value even if they collude with each other. Moreover, the vector space secret sharing supports multiple ways to reveal results (see Section 4.2 for details), i.e. different linear combinations of *public vectors* held by each party. Therefore, pMPL can tolerate that one of *assistant parties* drops out.

6.3 Complex Models in MPL Frameworks

pMPL supports various typical machine learning models, including linear regression, logistic regression, and BP neural networks, following current mainstream MPL frameworks. To further demonstrate the performance of pMPL, we conduct several experiments on more complex datasets, including Fashion-MNIST and SVHN. We compare the training accuracy of machine learning models trained with pMPL against the accuracy of machine learning models trained with plaintext data for the 10-class classification. As is shown in Appendix C, the results show that, under the same model structure, the accuracy of the machine learning models trained with pMPL is almost the same as that from the training data in plaintext.

For more complex and practical models, i.e. convolutional neural networks (CNN for short), as Max pooling, which is a key component of CNN, has no efficient secure computation protocol still now, we do not evaluate it in this paper. However, pMPL now has the potential to support CNN because pMPL has supported the key components of CNN, including full-connection layer, activation functions, and convolution operation that is essentially matrix multiplication. In future, we will optimize the secure computation protocol of Max pooling to support CNN models.

6.4 Comparison with Federated Learning

Typical federated learning frameworks [18, 19] also follow a hierarchical architecture, which has one centralized server and several clients. More specifically, federated learning iteratively executes the three steps as follows: (1) the centralized server sends the current global model to the clients or a subset of them; (2) each client tunes the global model received from the centralized server with its local data and sends model updates to the centralized server; (3) the centralized server updates the global model with the local model updates from clients. In federated learning, each client utilizes its own plaintext data to train a local model, and the communication among parties is coordinated by a centralized server.

Even though pMPL and federated learning both follow the hierarchical architecture, the centralized server in federated learning plays a totally different role in the training. It should hold more privileges than the *privileged party* in pMPL. In pMPL, the training is performed on shares, and the communication among these parties are in shares too. Thus, no party can infer private information from the intermediate results due to the security guarantees, which is shown in Appendix B, of the underlying techniques. In contrast, in federated learning, the model updates exchanged between clients and the centralized server might contain much sensitive information, which might be leaked [23, 39] to the centralized server (i.e. the centralized server might get clients' raw data).

6.5 Future Work

In future, we will optimize the efficiency of pMPL through reducing the communication rounds of matrix multiplication with truncation and reducing the communication rounds of activation functions evaluation. Meanwhile, we will support more complex machine learning models, such as CNN.

7 CONCLUSION

In this paper, we propose pMPL, an MPL framework based on the vector space secret sharing. To the best of our knowledge, pMPL is the first academic work to support a *privileged party* in an MPL framework. pMPL guarantees that even if two *assistant parties* collude with each other, only the *privileged party* can obtain the final result. Furthermore, pMPL tolerates one of the two *assistant parties* dropping out during training. That is, pMPL protects the interests of the *privileged party* while improving the robustness of the framework. Finally, the experimental results show that the performance of pMPL is promising when we compare it with state-of-the-art MPL frameworks. Especially, for the linear regression, pMPL is $16\times$ faster than TF-encrypted and $5\times$ for logistic regression in the LAN setting. In the WAN setting, although pMPL is slower than both SecureML and TF-encrypted, the performance is still promising. Because pMPL requires more communication overhead to ensure both the security (i.e., defending the collusion of two assist parties) and robustness, two requirements from novel practical scenarios.

ACKNOWLEDGMENTS

This paper is supported by NSFC (No. U1836207, 62172100) and STCSM (No. 21511101600). We thank all anonymous reviewers for their insightful comments. Weili Han is the corresponding author.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 805–817. https://doi.org/10.1145/2976749.2978331

[2] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings (Lecture Notes in Computer Science)*, Joan Feigenbaum (Ed.), Vol. 576. Springer, 420–432. https://doi.org/10.1007/3-540-46766-1_34

[3] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings (Lecture Notes in Computer Science)*, Sushil Jajodia and Javier López (Eds.), Vol. 5283. Springer, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[4] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1175–1191. https://doi.org/10.1145/3133956.3133982

[5] Ernest F. Brickell. 1989. Some Ideal Secret Sharing Schemes. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings (Lecture Notes in Computer Science)*, Jean-Jacques Quisquater and Joos Vandewalle (Eds.), Vol. 434. Springer, 468–475. https://doi.org/10.1007/3-540-46885-4_45

[6] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2020. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 459–480. https://doi.org/10.2478/popets-2020-0036

[7] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/trident-efficient-4pc-framework-for-privacy-preserving-machine-learning/

[8] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X. Liu, Hao Wang, and Cheng Hong. 2021. When Homomorphic Encryption Marries Secret Sharing: Secure Large-Scale Sparse Logistic Regression and Applications in Risk Control. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, Feida Zhu, Beng Chin Ooi, and Chunyan Miao (Eds.). ACM, 2652–2662. https://doi.org/10.1145/3447548.3467210

[9] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. 2018. Private Machine Learning in TensorFlow using Secure Computation. *CoRR* abs/1810.08130 (2018). arXiv:1810.08130 http://arxiv.org/abs/1810.08130

[10] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2183–2200. https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov

[11] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1102–1120. https://doi.org/10.1109/SP.2019.00078

[12] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniele Micciancio and Thomas Ristenpart (Eds.), Vol. 12171. Springer, 823–852. https://doi.org/10.1007/978-3-030-56880-1_29

[13] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *nature* 542, 7639 (2017), 115–118.

[14] Rasool Fakoor, Faisal Ladhak, Azade Nazi, and Manfred Huber. 2013. Using deep learning to enhance cancer diagnosis and classification. In *Proceedings of the international conference on machine learning*, Vol. 28. ACM New York, USA.

[15] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. 2021. Sharpness-aware Minimization for Efficiently Improving Generalization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=6Tm1mposlrM

[16] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. 2018. Privacy-Preserving Ridge Regression with only Linearly-Homomorphic Encryption. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings (Lecture Notes in Computer Science)*, Bart Preneel and Frederik Vercauteren (Eds.), Vol. 10892. Springer, 243–261. https://doi.org/10.1007/978-3-319-93387-0_13

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 218–229. https://doi.org/10.1145/28395.28420

[18] Jakub Konečný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated Optimization: Distributed Machine Learning for On-Device Intelligence. *CoRR* abs/1610.02527 (2016). arXiv:1610.02527 http://arxiv.org/abs/1610.02527

[19] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated Learning: Strategies for Improving Communication Efficiency. *CoRR* abs/1610.05492 (2016). arXiv:1610.05492 http://arxiv.org/abs/1610.05492

[20] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2651–2668. https://www.usenix.org/conference/usenixsecurity21/presentation/koti

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[22] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. 2021. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I (Lecture Notes in Computer Science)*, Nikita Borisov and Claudia Díaz (Eds.), Vol. 12674. Springer, 249–270. https://doi.org/10.1007/978-3-662-64322-8_12

[23] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. 2019. Exploiting Unintended Feature Leakage in Collaborative Learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 691–706. https://doi.org/10.1109/SP.2019.00029

[24] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 35–52. https://doi.org/10.1145/3243734.3243760

[25] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 591–602. https://doi.org/10.1145/2810103.2813705

[26] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 19–38. https://doi.org/10.1109/SP.2017.12

[27] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. (2011).

[28] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 619–636. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[29] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. Deepsecure: scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2:1–2:6. https://doi.org/10.1145/3195970.3196023

[30] Wenqiang Ruan, Mingxin Xu, Haoyang Jia, Zhenhuan Wu, Lushan Song, and Weili Han. 2021. Privacy Compliance: Can Technology Come to the Rescue? *IEEE Secur. Priv.* 19, 4 (2021), 37–43. https://doi.org/10.1109/MSEC.2021.3078218

[31] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[32] Lushan Song, Haoqi Wu, Wenqiang Ruan, and Weili Han. 2020. SoK: Training Machine Learning Models over Multiple Sources with Privacy Preservation. *CoRR* abs/2012.03386 (2020). arXiv:2012.03386 https://arxiv.org/abs/2012.03386

[33] Paul Voigt and Axel Von dem Bussche. 2017. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* (2017).

[34] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49. https://doi.org/10.2478/popets-2019-0035

[35] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[36] Guowen Xu, Hongwei Li, Sen Liu, Kan Yang, and Xiaodong Lin. 2020. VerifyNet: Secure and Verifiable Federated Learning. *IEEE Trans. Inf. Forensics Secur.* 15 (2020), 911–926. https://doi.org/10.1109/TIFS.2019.2929409

[37] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982.* IEEE Computer Society, 160–164. https://doi.org/10.1109/SFCS.1982.38

[38] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986.* IEEE Computer Society, 162–167. https://doi.org/10.1109/SFCS.1986.25

[39] Ligeng Zhu and Song Han. 2020. Deep Leakage from Gradients. In *Federated Learning - Privacy and Incentive*, Qiang Yang, Lixin Fan, and Han Yu (Eds.). Lecture Notes in Computer Science, Vol. 12500. Springer, 17–31. https://doi.org/10.1007/978-3-030-63076-8_2

[40] Youwen Zhu and Tsuyoshi Takagi. 2015. Efficient scalar product protocol and its privacy-preserving application. *Int. J. Electron. Secur. Digit. Forensics* 7, 1 (2015), 1–19. https://doi.org/10.1504/IJESDF.2015.067985

A SHARES HELD BY EACH PARTY

A.1 Shares During Secure Multiplication

We show the shares held by each party P_i during the execution of secure multiplication protocol $\prod_{\text{mul}}(\mathcal{P}, \langle x\rangle, \langle y\rangle)$ (Protocol 3) in Table 7. More specifically, for the first line, each party P_i holds $\langle u\rangle_i, \langle v\rangle_i, \langle h\rangle_i$ by performing $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4) during the offline phase. P_3 additionally holds $\langle u\rangle_3$, $\langle v\rangle_3$, $\langle h\rangle_3$. The second line in Table 7 shows the shares of two inputs x and y held by each party P_i. For the rest three lines, they are corresponding to the three steps of $\prod_{\text{mul}}(\mathcal{P}, \langle x\rangle, \langle y\rangle)$ (Protocol 3).

A.2 Shares During Vector Multiplication Triplets Generation

We show the shares held by each party P_i during the execution of vector multiplication triplet generation protocol $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4) in Table 8. More specifically, the three steps of generating $\langle u\rangle_i$, $\langle v\rangle_i$ is corresponding to the first three lines of Table 8. For the four steps of generating $\langle h\rangle_i$, it is corresponding to the last four lines of Table 8.

B SECURITY OF OUR DESIGNS

In this section, we introduce the security of our design using the standard real/ideal world paradigm. We use $\mathcal{S}$ to denote an ideal-world static adversary (simulator) for a real-world adversary . $\mathcal{S}$ acts as the honest parties and simulates the messages received by real-world adversary during the protocol. For each of the constructions, we provide the simulation proof for the case of corrupt of P_0 and the case of corrupt P_1 and P_2 (i.e. P_1 and P_2 collude with each other).

Sharing Protocol: The ideal functionality $\mathcal{F}_{\text{shr}}$ realising sharing protocol $\prod_{\text{shr}}(P_i, x)$ (Protocol 1) is presented in Figure 4. Here we assume that P_0 inputs x.

Theorem 1. *Sharing protocol $\prod_{\text{shr}}(P_i, x)$ (Protocol 1) securely realizes the functionality $\mathcal{F}_{\text{shr}}$ (Functionality 4) in the presence of static semi-honest adversary.*

Functionality $\mathcal{F}_{\text{shr}}$
Input:

- P_0 inputs x.

Output:

- P_0 outputs $\langle x\rangle_0$ and $\langle x\rangle_3$;
- P_1 outputs $\langle x\rangle_1$;
- P_2 outputs $\langle x\rangle_2$.

Figure 4: Functionality $\mathcal{F}_{\text{shr}}$

Proof: We present the simulation for the case for corrupt P_0 and the case for corrupt P_1 and P_2 as shown in Figure 5 and Figure 6 respectively.

Simulator $\mathcal{S}_{\text{shr}}^{\text{P}_0}$

```
$\mathcal{S}_{\text{shr}}^{\text{P}_0}$ receives $x$ and $\Phi(\mathcal{P})$ from $P_0$.
$\mathcal{S}_{\text{shr}}^{\text{P}_0}$ selects two random values $s_1, s_2$, and constructs a vector $\vec{s} = (x, s_1, s_2)^T$.
$\mathcal{S}_{\text{shr}}^{\text{P}_0}$ computes
    $\langle x\rangle_0 = \Phi(0) \times \vec{s}, \quad \langle x\rangle_1 = \Phi(1) \times \vec{s}$
    $\langle x\rangle_2 = \Phi(2) \times \vec{s}, \quad \langle x\rangle_3 = \Phi(3) \times \vec{s}$
$\mathcal{S}_{\text{shr}}^{\text{P}_0}$ outputs $(x, \langle x\rangle_0, \langle x\rangle_1, \langle x\rangle_2, \langle x\rangle_3)$.
```

Figure 5: Simulator $\mathcal{S}_{\text{shr}}^{\text{P}_0}$

Simulator $\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$

```
$\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$ receives $\Phi(\mathcal{P})$ from $P_1$.
$\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$ selects three random values $x, s_1, s_2$, and constructs a vector $\vec{s} = (x, s_1, s_2)^T$.
$\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$ computes
    $\langle x\rangle_1 = \Phi(1) \times \vec{s}, \ \langle x\rangle_2 = \Phi(2) \times \vec{s}$
$\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$ outputs $(\langle x\rangle_1, \langle x\rangle_2)$.
```

Figure 6: Simulator $\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$

We denote $\textbf{view}_{P_0}^{shr}$ and $\textbf{view}_{P_1,P_2}^{shr}$ as the views of P_0 and P_1, P_2 respectively. We note that P_0's view and $\mathcal{S}_{\text{shr}}^{\text{P}_0}$'s output are identical, the probability distribution of P_1 and P_2's views and $\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}$'s output are identical. Therefore we have the following equations:

$$\mathcal{S}_{\text{shr}}^{\text{P}_0}(x, \langle x\rangle_0, \langle x\rangle_3) \cong \textbf{view}_{P_0}^{shr}(x, \langle x\rangle_k, k \in \{0, 1, 2, 3\})$$
$$\mathcal{S}_{\text{shr}}^{\text{P}_1,\text{P}_2}(\emptyset, \langle x\rangle_1, \langle x\rangle_2) \cong \textbf{view}_{P_1,P_2}^{shr}(x, \langle x\rangle_k, k \in \{0, 1, 2, 3\})$$

Reconstruction Protocol: The ideal functionality $\mathcal{F}_{\text{rec}}$ realising reconstruction protocol $\prod_{\text{rec}}(\mathcal{P}, \langle x\rangle)$ (Protocol 2) is presented in Figure 7. Here, we only consider the case of no party drops out.

Theorem 2. *Reconstruction protocol $\prod_{\text{rec}}(P_i, \langle x\rangle)$ (Protocol 2) securely realizes the functionality $\mathcal{F}_{\text{rec}}$ (Figure 7) in the presence of static semi-honest adversary.*

Table 7: Shares held by each party during the execution of $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ (Protocol 3). For each line, the shares held by each party P_i correspond to each step in $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ (Protocol 3).

Step	*Privileged party* P_0	*Assistant party* P_1	*Assistant party* P_2
Pre-generating	$\langle u \rangle_0, \langle u \rangle_3, \langle v \rangle_0, \langle v \rangle_3, \langle h \rangle_0, \langle h \rangle_3$	$\langle u \rangle_1, \langle v \rangle_1, \langle h \rangle_1$	$\langle u \rangle_2, \langle v \rangle_2, \langle h \rangle_2$
Inputting	$\langle x \rangle_0, \langle x \rangle_3, \langle y \rangle_0, \langle y \rangle_3$	$\langle x \rangle_1, \langle y \rangle_1$	$\langle x \rangle_2, \langle y \rangle_2$
Locally computing	$\langle e \rangle_0 = \langle x \rangle_0 + \langle u \rangle_0$ $\langle e \rangle_3 = \langle x \rangle_3 + \langle u \rangle_3$ $\langle f \rangle_0 = \langle y \rangle_0 + \langle v \rangle_0$ $\langle f \rangle_3 = \langle y \rangle_3 + \langle v \rangle_3$	$\langle e \rangle_1 = \langle x \rangle_1 + \langle u \rangle_1$ $\langle f \rangle_1 = \langle y \rangle_1 + \langle v \rangle_1$	$\langle e \rangle_2 = \langle x \rangle_2 + \langle u \rangle_2$ $\langle f \rangle_2 = \langle y \rangle_2 + \langle v \rangle_2$
Communicating	$\prod_{\text{rec}}(\mathcal{P}, \langle e \rangle)$ and $\prod_{\text{rec}}(\mathcal{P}, \langle f \rangle)$		
Locally computing	$\langle z \rangle_0 = \langle x \rangle_0 \cdot f - \langle v \rangle_0 \cdot e + \langle h \rangle_0$ $\langle z \rangle_3 = \langle x \rangle_3 \cdot f - \langle v \rangle_3 \cdot e + \langle h \rangle_3$	$\langle z \rangle_1 = \langle x \rangle_1 \cdot f - \langle v \rangle_1 \cdot e + \langle h \rangle_1$	$\langle z \rangle_2 = \langle x \rangle_2 \cdot f - \langle v \rangle_i \cdot e + \langle h \rangle_2$

Table 8: Shares held by each party during the execution of $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4). For each line, the shares held by each party correspond to each step in $\prod_{\text{vmtgen}}(\mathcal{P})$ (Protocol 4).

Step	*Privileged party* P_0	*Assistant party* P_1	*Assistant party* P_2
Generating random values	two random values u_0, v_0	two random values u_1, v_1	two random values u_2, v_2
Executing $\prod_{\text{shr}}(P_i, u_i)$ and $\prod_{\text{shr}}(P_i, v_i)$	$\langle u_0 \rangle_0, \langle u_1 \rangle_0, \langle u_2 \rangle_0$ $\langle v_0 \rangle_0, \langle v_1 \rangle_0, \langle v_2 \rangle_0$ $\langle u_0 \rangle_3, \langle u_1 \rangle_3, \langle u_2 \rangle_3$ $\langle v_0 \rangle_3, \langle v_1 \rangle_3, \langle v_2 \rangle_3$	$\langle u_0 \rangle_1, \langle u_1 \rangle_1, \langle u_2 \rangle_1$ $\langle v_0 \rangle_1, \langle v_1 \rangle_1, \langle v_2 \rangle_1$	$\langle u_0 \rangle_2, \langle u_1 \rangle_2, \langle u_2 \rangle_2$ $\langle v_0 \rangle_2, \langle v_1 \rangle_2, \langle v_2 \rangle_2$
Locally computing	$\langle u \rangle_0 = \langle u_0 \rangle_0 + \langle u_1 \rangle_0 + \langle u_2 \rangle_0$ $\langle v \rangle_0 = \langle v_0 \rangle_0 + \langle v_1 \rangle_0 + \langle v_2 \rangle_0$ $\langle u \rangle_3 = \langle u_0 \rangle_3 + \langle u_1 \rangle_3 + \langle u_2 \rangle_3$ $\langle v \rangle_3 = \langle v_0 \rangle_3 + \langle v_1 \rangle_3 + \langle v_2 \rangle_3$	$\langle u \rangle_1 = \langle u_0 \rangle_1 + \langle u_1 \rangle_1 + \langle u_2 \rangle_1$ $\langle v \rangle_1 = \langle v_0 \rangle_1 + \langle v_1 \rangle_1 + \langle v_2 \rangle_1$	$\langle u \rangle_2 = \langle u_0 \rangle_2 + \langle u_1 \rangle_2 + \langle u_2 \rangle_2$ $\langle v \rangle_2 = \langle v_0 \rangle_2 + \langle v_1 \rangle_2 + \langle v_2 \rangle_2$
Secure computing	$[u_0 * v_1 + v_0 * u_1]_0$ $[u_0 * v_2 + v_0 * u_2]_0$	$[u_0 * v_1 + v_0 * u_1]_1$ $[u_1 * v_2 + v_1 * u_2]_1$	$[u_0 * v_2 + v_0 * u_2]_2$ $[u_1 * v_2 + v_1 * u_2]_2$
Locally computing	$h_0 = u_0 * v_0 + [u_0 * v_1 + v_0 * u_1]_0$ $+[u_0 * v_2 + v_0 * u_2]_0$	$h_1 = u_1 * v_1 + [u_0 * v_1 + v_0 * u_1]_1$ $+[u_1 * v_2 + v_1 * u_2]_1$	$h_2 = u_2 * v_2 + [u_0 * v_2 + v_0 * u_2]_2$ $+[u_1 * v_2 + v_1 * u_2]_2$
Executing $\prod_{\text{shr}}(P_i, h_i)$	$\langle h_0 \rangle_0, \langle h_1 \rangle_0, \langle h_2 \rangle_0$ $\langle h_0 \rangle_3, \langle h_1 \rangle_3, \langle h_2 \rangle_3$	$\langle h_0 \rangle_1, \langle h_1 \rangle_1, \langle h_2 \rangle_1$	$\langle h_0 \rangle_2, \langle h_1 \rangle_2, \langle h_2 \rangle_2$
Locally computing	$\langle h \rangle_0 = \langle h_0 \rangle_0 + \langle h_1 \rangle_0 + \langle h_2 \rangle_0$ $\langle h \rangle_3 = \langle h_0 \rangle_3 + \langle h_1 \rangle_3 + \langle h_2 \rangle_3$	$\langle h \rangle_1 = \langle h_0 \rangle_1 + \langle h_1 \rangle_1 + \langle h_2 \rangle_1$	$\langle h \rangle_2 = \langle h_0 \rangle_2 + \langle h_1 \rangle_2 + \langle h_2 \rangle_2$

Functionality $\mathcal{F}_{\text{rec}}$
Input:

- P_0 inputs $\langle x \rangle_0$;
- P_1 inputs $\langle x \rangle_1$;
- P_2 inputs $\langle x \rangle_2$.

Output:

- P_0, P_1 and P_2 all output x.

Figure 7: Functionality $\mathcal{F}_{\text{rec}}$

Proof: We present the simulation for the case for corrupt P_0 and the case for corrupt P_1 and P_2 as shown in Figure 8 and Figure 9 respectively.

We denote $\textbf{view}_{P_0}^{rec}$ and $\textbf{view}_{P_1,P_2}^{rec}$ as the views of P_0 and P_1, P_2 respectively. We note that the probability distribution of P_0's view and $\mathcal{S}_{\text{rec}}^{\text{P}_0}$'s output are identical, the probability distribution of P_1 and P_2's views and $\mathcal{S}_{\text{rec}}^{\text{P}_1,\text{P}_2}$'s output are identical. Therefore we have the following equations:

$$\mathcal{S}_{\text{rec}}^{\text{P}_0}(\langle x \rangle_0, x) \cong \textbf{view}_{P_0}^{rec}(\langle x \rangle_0, \langle x \rangle_1, \langle x \rangle_2, x)$$
$$\mathcal{S}_{\text{rec}}^{\text{P}_1,\text{P}_2}(\langle x \rangle_1, \langle x \rangle_2, x) \cong \textbf{view}_{P_1,P_2}^{rec}(\langle x \rangle_0, \langle x \rangle_1, \langle x \rangle_2, x)$$

Simulator $\mathcal{S}_{\text{rec}}^{\text{P}_0}$

```
1: $\mathcal{S}_{\text{rec}}^{\text{P}_0}$ receives $\langle x \rangle_0$ and $c_0, c_1, c_2$ from $P_0$.
2: $\mathcal{S}_{\text{rec}}^{\text{P}_0}$ selects two random values $\langle x \rangle_1, \langle x \rangle_2$.
3: $\mathcal{S}_{\text{rec}}^{\text{P}_0}$ computes
       $x = c_0 \cdot \langle x \rangle_0 + c_1 \cdot \langle x \rangle_1 + c_2 \cdot \langle x \rangle_2$
4: $\mathcal{S}_{\text{rec}}^{\text{P}_0}$ outputs $(\langle x \rangle_0, \langle x \rangle_1, \langle x \rangle_2, x)$.
```

Figure 8: Simulator $\mathcal{S}_{\text{rec}}^{\text{P}_1,\text{P}_2}$

Multiplication Protocol: The ideal functionality $\mathcal{F}_{\text{mul}}$ realising multiplication protocol $\prod_{\text{mul}}(\mathcal{P}, \langle x \rangle, \langle y \rangle)$ (Protocol 3) is presented in Figure 10.

```
Simulator $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$
  $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$ receives $\langle x\rangle_1, \langle x\rangle_2$ and $c_0, c_1, c_2$ from $P_1, P_2$.
  $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$ selects a random value $\langle x\rangle_0$.
  $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$ computes
      $x = c_0 \cdot \langle x\rangle_0 + c_1 \cdot \langle x\rangle_1 + c_2 \cdot \langle x\rangle_2$
  $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$ outputs $(\langle x\rangle_0, \langle x\rangle_1, \langle x\rangle_2, x)$.
```

Figure 9: Simulator $\mathcal{S}_{\mathrm{rec}}^{\mathrm{P_1,P_2}}$

Functionality $\mathcal{F}_{\mathrm{mul}}$
Input:

- P_0 inputs $\langle x\rangle_0, \langle y\rangle_0$ and $\langle x\rangle_3, \langle y\rangle_3$;
- P_1 inputs $\langle x\rangle_1, \langle y\rangle_1$;
- P_2 inputs $\langle x\rangle_2, \langle y\rangle_2$.

Output:

- P_0 outputs $\langle z\rangle_0$ and $\langle z\rangle_1$;
- P_1 outputs $\langle z\rangle_1$;
- P_2 outputs $\langle z\rangle_2$, where $z = x \cdot y$.

Figure 10: Functionality $\mathcal{F}_{\mathrm{mul}}$

Theorem 3. *Multiplication protocol $\prod_{\mathrm{mul}}(\mathcal{P}, \langle x\rangle, \langle y\rangle)$(Protocol 3) securely realizes the functionality $\mathcal{F}_{\mathrm{mul}}$ (Figure 10) in the presence of static semi-honest adversary.*

Proof: We present the simulation for the case for corrupt P_0 and the case for corrupt P_1 and P_2 as shown in Figure 11 and Figure 12 respectively.

```
Simulator $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ receives $\langle x\rangle_0, \langle y\rangle_0, \langle x\rangle_3, \langle y\rangle_3$ from $P_0$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ receives $\langle u\rangle_0, \langle v\rangle_0, \langle h\rangle_0, \langle u\rangle_3, \langle v\rangle_3, \langle h\rangle_3$ from $P_0$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ computes
      $\langle e\rangle_0 = \langle x\rangle_0 + \langle u\rangle_0, \quad \langle f\rangle_0 = \langle y\rangle_0 + \langle v\rangle_0$
      $\langle e\rangle_3 = \langle x\rangle_3 + \langle u\rangle_3, \quad \langle f\rangle_3 = \langle y\rangle_3 + \langle v\rangle_3$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ selects random values $\langle e\rangle_1, \langle f\rangle_1, \langle e\rangle_2, \langle f\rangle_2$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ computes
      $e = c_0 \cdot \langle e\rangle_0 + c_1 \cdot \langle e\rangle_1 + c_2 \cdot \langle e\rangle_2$
      $f = c_0 \cdot \langle f\rangle_0 + c_1 \cdot \langle f\rangle_1 + c_2 \cdot \langle f\rangle_2$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ computes
      $\langle z\rangle_0 = \langle x\rangle_0 \cdot f - \langle v\rangle_0 \cdot e + \langle h\rangle_0$
      $\langle z\rangle_3 = \langle x\rangle_3 \cdot f - \langle v\rangle_3 \cdot e + \langle h\rangle_3$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$ outputs $(\langle x\rangle_0, \langle x\rangle_3, \langle e\rangle_j, \langle f\rangle_j, \langle z\rangle_0, \langle z\rangle_3, j \in \{1, 2\})$.
```

Figure 11: Simulator $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$

```
Simulator$\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ receives $\langle x\rangle_1, \langle y\rangle_1, \langle x\rangle_2, \langle y\rangle_2$ from $P_1, P_2$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ receives $\langle u\rangle_1, \langle v\rangle_1, \langle h\rangle_1, \langle u\rangle_2, \langle v\rangle_2, \langle h\rangle_2$ from $P_1, P_2$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ computes
      $\langle e\rangle_1 = \langle x\rangle_1 + \langle u\rangle_1, \quad \langle f\rangle_1 = \langle y\rangle_1 + \langle v\rangle_1$
      $\langle e\rangle_2 = \langle x\rangle_2 + \langle u\rangle_2, \quad \langle f\rangle_2 = \langle y\rangle_2 + \langle v\rangle_2$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ selects random values $\langle e\rangle_0, \langle f\rangle_0$.
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ computes
      $e = c_0 \cdot \langle e\rangle_0 + c_1 \cdot \langle e\rangle_1 + c_2 \cdot \langle e\rangle_2$
      $f = c_0 \cdot \langle f\rangle_0 + c_1 \cdot \langle f\rangle_1 + c_2 \cdot \langle f\rangle_2$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ computes
      $\langle z\rangle_1 = \langle x\rangle_1 \cdot f - \langle v\rangle_1 \cdot e + \langle h\rangle_1$
      $\langle z\rangle_2 = \langle x\rangle_2 \cdot f - \langle v\rangle_2 \cdot e + \langle h\rangle_2$
  $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$ outputs $(\langle x\rangle_j, \langle e\rangle_0, \langle f\rangle_0, \langle z\rangle_j, j \in \{1, 2\})$.
```

Figure 12: Simulator $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}$

We denote $\mathbf{view}_{P_0}^{mul}$ and $\mathbf{view}_{P_1,P_2}^{mul}$ as the views of P_0 and P_1, P_2 respectively. We note that the probability distribution of P_0's view and $\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}$'s output are identical, P_1 and P_2's view and $\mathcal{S}_{\mathrm{shr}}^{\mathrm{P_1,P_2}}$'s output are identical. Therefore we have the following equations:

$$\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_0}}(\langle x\rangle_0, \langle y\rangle_0, \langle x\rangle_3, \langle y\rangle_3, \langle z\rangle_0, \langle z\rangle_3) \cong$$
$$\mathbf{view}_{P_0}^{mul}(\langle x\rangle_k, \langle y\rangle_k, \langle z\rangle_k, k \in \{0, 1, 2, 3\})$$
$$\mathcal{S}_{\mathrm{mul}}^{\mathrm{P_1,P_2}}(\langle x\rangle_1, \langle y\rangle_1, \langle x\rangle_2, \langle y\rangle_2, \langle z\rangle_1, \langle z\rangle_2) \cong$$
$$\mathbf{view}_{P_1,P_2}^{mul}(\langle x\rangle_k, \langle y\rangle_k, \langle z\rangle_k, k \in \{0, 1, 2, 3\})$$

Sharing conversion Protocol: Here, we only analyze the security of protocol $\prod_{\mathrm{a2v}}(\mathcal{P}, [x])$ (Protocol 5) since protocol $\prod_{\mathrm{v2a}}(\mathcal{P}, \langle x\rangle)$ is executed locally. The ideal functionality $\mathcal{F}_{\mathrm{a2v}}$ realising protocol $\prod_{\mathrm{a2v}}(\mathcal{P}, [x])$ (Protocol 5) is presented in Figure 13.

Functionality $\mathcal{F}_{\mathrm{a2v}}$
Input:

- P_0 inputs $[x]_0$;
- P_1 inputs $[x]_1$;
- P_2 inputs $[x]_2$.

Output:

- P_0 outputs $\langle x\rangle_0$ and $\langle x\rangle_3$;
- P_1 outputs $\langle x\rangle_1$;
- P_2 outputs $\langle x\rangle_2$.

Figure 13: Functionality $\mathcal{F}_{\mathrm{a2v}}$

Theorem 4. *Sharing conversion protocol $\prod_{\mathrm{a2v}}(\mathcal{P}, [x]$ (Protocol 5) securely realizes the functionality $\mathcal{F}_{\mathrm{a2v}}$ (Figure 13) in the presence of static semi-honest adversary.*

Proof: We present the simulation for the case for corrupt P_0 and the case for corrupt P_1 and P_2 as shown in Figure 14 and Figure 15 respectively.

```
**Simulator**$\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$ receives $[x]_0$, $[x]_3$ and $c_0, a_1, a_2, a_3, \langle k \rangle_0$ from $P_0$.
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$ selects random values $\langle x+k \rangle_1$,$\langle x+k \rangle_2$,
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$ computes
      $\langle x \rangle_0 = [x]_0 / c_0$
      $\langle x \rangle_3 = a_1 \cdot \langle x+k \rangle_1 + a_2 \cdot \langle x+k \rangle_2 - \langle k \rangle_3$
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$ outputs $([x]_0,\ [x]_3,\ \langle x+k \rangle_1,\ \langle x+k \rangle_2,\ \langle x \rangle_0,\ \langle x \rangle_3)$.
```

Figure 14: Simulator $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$

```
**Simulator**$\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$ receives $[x]_1, [x]_2$ and $c_1, c_2, \langle k \rangle_1, \langle k \rangle_2$ from $P_1, P_2$.
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$ computes
      $\langle x \rangle_1 = [x]_1 / c_1,\ \langle x \rangle_2 = [x]_2 / c_2$
      $\langle x+k \rangle_1 = \langle x \rangle_1 + \langle k \rangle_1,\ \langle x+k \rangle_2 = \langle x \rangle_2 + \langle k \rangle_2$
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$ outputs $([x]_1, [x]_2, \langle x \rangle_1, \langle x \rangle_2)$.
```

Figure 15: Simulator $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$

We denote $\mathbf{view}^{a2v}_{P_0}$ and $\mathbf{view}^{a2v}_{P_1,P_2}$ as the views of P_0 and P_1, P_2 respectively. We note that the probability distribution of P_0's view and $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}$'s output are identical, P_1 and P_2's view and $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}$'s output are identical. Therefore we have the following equations:

$$\mathcal{S}^{\mathrm{P}_0}_{\mathrm{a2v}}([x]_0, [x]_3, \langle x \rangle_0, \langle x \rangle_3) \cong \mathbf{view}^{a2v}_{P_0}([x]_i, \langle x \rangle_k, k \in \{0,1,2,3\})$$

$$\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{a2v}}([x]_1, [x]_2, \langle x \rangle_1, \langle x \rangle_2) \cong \mathbf{view}^{a2v}_{P_1,P_2}([x]_i, \langle x \rangle_k, k \in \{0,1,2,3\})$$

Truncation Protocol: The ideal functionality $\mathcal{F}_{\mathrm{trunc}}$ realizing truncation protocol $\prod_{\mathrm{trunc}}(\mathcal{P}, \langle z \rangle)$ (Protocol 6) is presented in Figure 16.

```
**Functionality** $\mathcal{F}_{\mathrm{trunc}}$
**Input:**
    • $P_0$ inputs $\langle z \rangle_0$;
    • $P_1$ inputs $\langle z \rangle_1$;
    • $P_2$ inputs $\langle z \rangle_2$.
**Output:**
    • $P_0$ outputs $\langle z' \rangle_0$ and $\langle z' \rangle_3$;
    • $P_1$ outputs $\langle z' \rangle_1$;
    • $P_2$ outputs $\langle z' \rangle_2$, where $z' = z/2^{\ell_f}$.
```

Figure 16: Functionality $\mathcal{F}_{\mathrm{trunc}}$

Theorem 5. *Truncation protocol $\prod_{\mathrm{trunc}}(\mathcal{P}, \langle z \rangle$ (Protocol 6) securely realizes the functionality $\mathcal{F}_{\mathrm{trunc}}$ (Functionality 16) in the presence of static semi-honest adversary.*

Proof: We present the simulation for the case for corrupt P_0 and the case for corrupt P_1 and P_2 as shown in Figure 17 and Figure 18 respectively.

```
**Simulator** $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$ receives $\langle z \rangle_0$ and $c_0, c_1, c_2, \langle r \rangle_0, \langle r' \rangle_0, \langle r' \rangle_3$ from $P_0$.
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$ selects random values $\langle z-r \rangle_1, \langle z-r \rangle_2$.
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$ computes
      $z - r = c_0 \cdot (\langle z \rangle_0 - \langle r \rangle_0) + c_1 \cdot \langle z-r \rangle_1 + c_2 \cdot \langle z-r \rangle_2$
      $\langle z' \rangle_0 = (z-r)/(2^{\ell_f} \cdot c_0) + \langle r' \rangle_0$
      $\langle z' \rangle_3 = \langle r' \rangle_3$
  $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$ outputs $(\langle z \rangle_0, \langle z-r \rangle_1, \langle z-r \rangle_2, \langle z' \rangle_0, \langle z' \rangle_3)$.
```

Figure 17: Simulator $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$

```
**Simulator**$\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$ receives $\langle r' \rangle_1, \langle r' \rangle_2$ from $P_1, P_2$.
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$ computes
      $\langle z-r \rangle_1 = \langle z \rangle_1 - \langle r \rangle_1\ \langle z-r \rangle_2 = \langle z \rangle_2 - \langle r \rangle_2$
      $\langle z' \rangle_1 = \langle r' \rangle_1\ \langle z' \rangle_2 = \langle r' \rangle_2$
  $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$ outputs $(\langle z \rangle_1, \langle z \rangle_2, \langle z' \rangle_1, \langle z' \rangle_2)$.
```

Figure 18: Simulator $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$

We denote $\mathbf{view}^{trunc}_{P_0}$ and $\mathbf{view}^{trunc}_{P_1,P_2}$ as the views of P_0 and P_1, P_2 respectively. We note that the probability distribution of P_0's view and $\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}$'s output are identical, P_1 and P_2's view and $\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}$'s output are identical. Therefore we have the following equations:

$$\mathcal{S}^{\mathrm{P}_0}_{\mathrm{trunc}}(\langle z \rangle_0, \langle z \rangle_3, \langle z' \rangle_0, \langle z' \rangle_3) \cong \mathbf{view}^{trunc}_{P_0}(\langle z \rangle_k, \langle z' \rangle_k, k \in \{0,1,2,3\})$$

$$\mathcal{S}^{\mathrm{P}_1,\mathrm{P}_2}_{\mathrm{trunc}}(\langle z \rangle_1, \langle z \rangle_2, \langle z' \rangle_1, \langle z' \rangle_2) \cong \mathbf{view}^{trunc}_{P_1,P_2}(\langle z \rangle_k, \langle z' \rangle_k, k \in \{0,1,2,3\})$$

C ACCURACY EVALUATION OVER MORE COMPLEX DATASETS

We evaluate the accuracy of typical machine learning models, including linear regression, logistic regression, and BP neural networks, trained with pMPL on more complex datasets, which are Fashion-MNIST and SVHN. (1) Fashion-MNIST is a dataset similar to MNIST. It also contains 60,000 training samples and 10,000 test samples. Each sample is a 28 × 28 grayscale image. Rather than handwritten digits as MNIST, Fashion-MNIST contains image samples of ten classes of clothing. (2) SVHN is a dataset from house numbers in Google Street View images. It incorporates more samples, i.e. 73,257 training samples and 26,032 test samples. Besides, each sample is a 32 × 32 RGB image, associated with a label from ten classes. Furthermore, lots of the images contain some distractors at the sides. Therefore, SVHN and Fashion-MNIST are both harder to classify than MNIST. The basic information of these datasets is shown in Table 9.

We conduct a series of experiments to compare the accuracy of machine learning models trained with pMPL and models trained with plaintext decimal data. As is shown in Table 10, the experimental

Table 9: Brief description of datasets used in pMPL.

Dataset	Fetures	Training samples	Test samples
MNIST	784	60,000	10,000
Fashion-MNIST	784	60,000	10,000
SVHN	3,072	73,257	26,032

results show that the accuracy of the machine learning models trained with pMPL is almost the same as those trained from the data in plaintext. Note that the accuracy of the models of linear regression and logistic regression on SVHN is very poor (about 20% both in pMPL and plaintext), thus not shown in Table 10. In addition, the accuracy of BP neural networks on SVHN is about 73%, much lower than the result (about 99% [15]) from the state-of-the-art neural networks. Thus, we argue that although pMPL presents a feasible framework with a privileged party, we should pay much attention to enabling pMPL to efficiently support the state-of-the-art deep neural networks in future.

Table 10: Accuracy of the typical machine learning models trained with pMPL (in secret shares) compared to the ones trained from the decimal data in plaintext.

Model	Dataset	Accuaracy	
		pMPL	Plaintext
Linear regression	MNIST	85.77%	85.80%
	Fashion-MNIST	80.69%	80.80%
Logistic regression	MNIST	91.07%	91.38%
	Fashion-MNIST	83.99%	84.01%
BP neural networks	MNIST	96.41%	96.52%
	Fashion-MNIST	86.47%	86.78%
	SVHN	73.31%	73.35%