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

Lushan Song
Fudan University
19110240022@fudan.edu.cn

Jiaxuan Wang
Fudan University
2021010090@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

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.

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.
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 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 secure joint 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 more 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. Wagw et al. [34] designed a three-party MPL framework SecureETN 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.

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 $t \leq n$ parties. That is, these frameworks follow the peer-to-peer architecture.

Figure 1: Practical scenarios
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. 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 them. Thus, we set each authorized set to include the privileged party mentioned above, and once training is completed, only assistant parties send their shares to the privileged party, while the privileged party does not send its shares to them. Therefore, 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.

**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.

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 \( P = \{ P_0, P_1, \ldots, P_n \} \) 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 \( P \), i.e. \( \Gamma \subseteq 2^P \). \( \Gamma \) is defined as an access structure on \( P \). Meanwhile, its element \( B_j \in \Gamma \) is defined as an authorized set 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 \( \Gamma \) cannot reveal the secret value. Then, with a large prime number \( X \) in linear aggression is to compute the output of function \( \phi \). The goal of linear regression is to find the coefficient parameters. The number of the epoch \( t \) is initialized 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 \( \hat{A}_i = f(U_i) \), where \( U_i = A_{i-1} \times W_i \), 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 \( E_m = (A_m - Y_m) \), and the error matrices of other layers are computed as \( E_i = (E_{i+1} \times W_i^T) \odot df(U_i) \), where \( \odot \) denotes the element-wise product, and \( df(\cdot) \) denotes the derivative of activation function of \( f(\cdot) \). After the backward propagation phase, we update the coefficient matrix as \( W_i := W_i - \alpha \Delta_{i-1} \times E_i \).

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 |
|--------|-------------|
| \( P \) | The set of parties |
| \( \Gamma \) | The access structure |
| \( B_j \) | The authorized set |
| \( \{ \} \) | The shares of additive secret sharing |
| \( \{ \}^2 \) | The shares of boolean sharing |
| \( \phi(\cdot) \) | The public matrix for vector space secret sharing |
| \( c_0, c_1, \ldots, c_d \) | The reconstruction coefficients |
| \( a_0, a_1 \) | The coefficients of the alternate vector |
| \( t \) | The number of bits to represent a fixed-point number |
| \( f \) | The number of bits to represent the fractional part of a fixed-point number |
| \( \langle u \rangle, \langle v \rangle \) | The vector multiplication triplet |
| \( B \) | The batch size |
| \( D \) | The dimension of the feature |
| \( E \) | The number of the epoch |

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 \( G(X) = f(X \times \hat{w}) \). The forward propagation stage in logistic regression is to compute \( f(X \times \hat{w}) \). Then, in the backward propagation stage, the coefficient parameters \( \hat{w} \) can be updated as:

\[
\hat{w} := \hat{w} - \alpha X^T (f(X \times \hat{w}) - Y)
\]
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 \( \text{pMPL} \), 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 \( P_0 \), even when \( P_1 \) and \( P_2 \) collude with each other. Furthermore, \( \text{pMPL} \) tolerates one assistant party \( (P_1 \text{ or } P_2) \) dropping out of training. As a result, the access structure \( \mathcal{F} \) in \( \text{pMPL} \) is \( \{\{P_0, P_1, P_2\}, \{P_0, P_1\}, \{P_0, P_2\}\} \).

3.1.2 Data representation. In machine learning, to train accurate 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 \( \text{pMPL} \), 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 \( P_0 \), even when \( P_1 \) and \( P_2 \) collude with each other. Furthermore, \( \text{pMPL} \) tolerates one assistant party \( (P_1 \text{ or } P_2) \) dropping out of training. As a result, the access structure \( \mathcal{F} \) in \( \text{pMPL} \) is \( \{\{P_0, P_1, P_2\}, \{P_0, P_1\}, \{P_0, P_2\}\} \).

3.2 Security Model

In this paper, we employ the semi-honest (also known as honest-but-curious or passive) security model in \( \text{pMPL} \). 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 that of the previous MPL frameworks (e.g. SecureML [26] and ABY3 [24]).

3.3 Robust Training

The robustness employed in \( \text{pMPL} \) ensures that training would continue even though one assistant party drops out. In \( \text{pMPL} \), an additional public 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 \( \text{pMPL} \)

In this section, we firstly introduce the sharing semantics of \( \text{pMPL} \), 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 \( \text{pMPL} \). Furthermore, we introduce the design of robustness of \( \text{pMPL} \). 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 \( x \in \mathbb{Z}_{2^t} \) 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 \( [x]_i \in \mathbb{Z}_{2^t} \) (\( i \in \{0, 1, 2\} \)), such that \( x = ([x]_0 + [x]_1 + [x]_2) \mod 2^t \), 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 \( \text{pMPL} \). 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 \( \Phi(P) \) as a 4x3 matrix. Here, for each party \( P_i \), the \( i \)-th row \( \Phi(i) \) of \( \Phi(P) \) 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 \( \Phi(P) \) 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 \((1,0,0) = c_0 \Phi(0) + c_1 \Phi(1) + c_2 \Phi(2)\).

- The public vector \( \Phi(3) \) can be represented linearly by the vectors \( \Phi(1) \) and \( \Phi(2) \), i.e., \( \Phi(3) = a_1 \Phi(1) + a_2 \Phi(2) \), 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 \( c'_{0}, c'_{1}, c'_{0}, c'_{1}, c'_{0}, c'_{1} \) such that \((1,0,0) = c'_0 \Phi(0) + c'_1 \Phi(1) + c'_0 \Phi(2) + c'_1 \Phi(3)\).

- 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 sets \( \{ \Phi(0), \Phi(3) \} \) and \( \{ \Phi(1), \Phi(2) \} \).

As pMPL performs the computations on the ring \( \mathbb{Z}_{2^r} \), both the values of public matrix \( \Phi(P) \) and reconstruction coefficients \( c_0, c_1, ..., c'_{3} \) should be elements of the ring \( \mathbb{Z}_{2^r} \).

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

\[
\begin{align*}
(1,0,0) &= c_0 \Phi(0) + c_1 \Phi(1) + c_2 \Phi(2) \\
&= c'_0 \Phi(0) + c'_1 \Phi(1) + c'_0 \Phi(2) + c'_1 \Phi(3) \\
&= c''_0 \Phi(0) + c''_2 \Phi(2) + c''_3 \Phi(3) \\
&= x \Phi(i) + \vec{s}.
\end{align*}
\]

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

**Protocol 1 \( \Pi_{shr}(P_i, x) \)**

**Input:** The secret value \( x \) held by \( P_i \)

**Output:** \( x \)

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. \( P_i \) sends \((x)\), where \( j \in \{1,2\} \). Meanwhile, \( P_i \) generates \((x) \) and \( (x) \).
3. If \( i \neq 3 \), \( P_i \) sends the alternate share \((x) \).

**Sharing Protocol:** As is shown in Protocol 1, \( \Pi_{shr}(P_i, x) \) enables \( P_i \) who holds the secret value \( x \) to generate \((x)\)-sharing of \( x \). In Step 1 of \( \Pi_{shr}(P_i, x) \), 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 \( \Pi_{shr}(P_i, x) \), Protocol 1, \( P_i \) considers two cases as follows:

- If \( P_i = P_0 \), \( P_i \) sends \((x)\) to two assistant parties \( P_j \) for \( j \in \{1,2\} \). Meanwhile, \( P_i \) generates \((x) \) as an alternate share \((x) \).

**Secure Addition:** Given two secret values \( x \) and \( y \), each party \( P_i \) holds shares \((x) \) and \( (y) \). To get the result of secure addition \((x + y)\), each party \( P_i \) can utilize the linearity property of the \((x)\)-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 ($x$ and $y$). 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 Beuver’s multiplication triple [2], to execute secure multiplication.

$$x \cdot y = x \cdot (y + v) - x \cdot v = x \cdot (y + v) - v \cdot (x + u - u)$$

$$= x \cdot (y + v) - v \cdot (x + u) + v \cdot u \quad (10)$$

Protocol 3 shows the secure multiplication protocol $\Pi_{mul}(P, (x, y))$ 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 $\Pi_{mul}(P, (x, y))$, we set $t = (u_1, u_2)^T$, $q = (q_1, q_2)^T$ uniformly random three-dimensional vectors and $\bar{t} = (h(t_1, t_2)^T = (u \cdot t_1, t_2)^T$, where $t_1, t_2$ are uniformly random values. We assume that all the parties have already shared vector multiplication triplet ($u$, $v$, $h$) in the offline phase. In the online phase of $\Pi_{mul}(P, (x, y))$, 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 $\Pi_{rec}(P, (e))$ (Protocol 2) and $\Pi_{rec}(P, (f))$ (Protocol 2). Finally, each party $P_i$ locally computes $\langle z \rangle_i = \langle x \rangle_i \cdot \langle f \rangle_i + \langle e \rangle_i \cdot h$. Similarly, $P_0$ additionally computes the alternate share $\langle z \rangle_3 = \langle x \rangle_3 \cdot \langle f \rangle_3 - \langle e \rangle_3 \cdot e + \langle h \rangle_3$. Protocol 3

**Preprocessing:** Parties pre-shared vector multiplication triplet ($u$, $v$, $h$) using $\Pi_{vmtgen}(P)$ (Protocol 4)

**Input:** ($x$) and ($y$)

**Output:** $\langle x \cdot y \rangle$

1. $P_1$ 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 $\Pi_{rec}(P, (e))$ (Protocol 2) and $\Pi_{rec}(P, (f))$ (Protocol 2).
3. $P_1$ locally computes $\langle z \rangle_i = \langle x \rangle_i \cdot \langle f \rangle_i - \langle e \rangle_i \cdot h$ and $P_0$ additionally computes the alternate share $\langle z \rangle_3 = \langle x \rangle_3 \cdot \langle f \rangle_3 - \langle e \rangle_3 \cdot e + \langle h \rangle_3$.

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

- **Generating ($u$, $v$) **and ($h$): As ($u$) and ($v$) are generated in the same way, we hereby take the generation of ($u$) as an example. Firstly, each party $P_i$ generates a random value $u_i$. Then they interactively execute $\Pi_{uni}(P_i, u_i)$ (Protocol 1). After that, each party $P_i$ holds three shares ($u_0(i)$, $u_1(i)$, $u_2(i)$). Besides, $P_0$ additionally holds another three alternate shares ($u_0(i)_3$, $u_1(i)_3$, $u_2(i)_3$). Then each party $P_i$ adds up these three shares locally to compute $\langle u \rangle_i = \langle u_0(i) \rangle + \langle u_1(i) \rangle + \langle u_2(i) \rangle$. $P_0$ additionally computes $\langle u \rangle_3 = \langle u_0(i)_3 \rangle + \langle u_1(i)_3 \rangle + \langle u_2(i)_3 \rangle$.

- **Generating ($h$):** Given shared random values ($u$) and ($v$) mentioned above, the key step of generating ($h$) is to compute the shares of their product. According to the process of generating ($u$) and ($v$), we can get that $u = u_0 + u_1 + u_2$ and $v = v_0 + v_1 + v_2$. Then:

$$h = uv = (u_0 + u_1 + u_2)(v_0 + v_1 + v_2) = u_0v_0 + u_0v_1 + u_0v_2 + u_1v_0 + u_1v_1 + u_1v_2 + u_2v_0 + u_2v_1 + u_2v_2 \quad (11)$$

where $u_0v_i (i \in \{0, 1, 2\})$ 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 $h_i = u_i v_i + [u_i v_{i+1} + u_{i+1} v_i] + [u_{i+1} v_{i-1} + u_{i-1} v_i]$. 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 $\Pi_{uni}(P_i, h_i)$ (Protocol 1) to get three shares ($h_0(i)$, $h_1(i)$), and ($h_2(i)$). $P_0$ additionally holds three alternate shares ($h_0(i)_3$, $h_1(i)_3$, and $h_2(i)_3$). At last, each party $P_i$ adds up the three shares locally to get $\langle h \rangle_i = \langle h_0(i) \rangle + \langle h_1(i) \rangle + \langle h_2(i) \rangle$. Besides, $P_0$ computes the alternate shares $\langle h \rangle_3$ and ($h_3$) in the same way.

**Generating ($h$):**

1. $P_0$ and $P_2$ interactively compute $[u_0(i) + u_1(i)]$, $P_0$ and $P_2$ interactively compute $[u_0(i) + u_2(i)]$. $P_1$ and $P_2$ interactively compute $[a_1(i) + a_2(i)]$.
2. $P_1$ locally computes $h_1 = u_2(i) + [u_2(i)_3 + u_1(i) + u_1(i)_3] + [u_1(i)_3 + u_1(i) + u_1(i)_3]$. $P_2$ locally computes $h_2 = u_2(i) + [u_0(i)_3 + u_0(i) + u_0(i)_3]$.
3. $P_2$ executes $\Pi_{uni}(P_2, h_2)$ (Protocol 1).
4. $P_1$ locally computes $h_1 = \langle h_1 \rangle_3 + \langle h_1 \rangle_3 + \langle h_1 \rangle_3$. $P_0$ additionally computes the alternate share $\langle h \rangle_3 = \langle h_0(i) \rangle + \langle h_1(i) \rangle + \langle h_2(i) \rangle$.

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

We hereby present two primitive modes of sharing conversion as follows:

- **Converting $\cdot$-shares to $[\cdot]$-shares:** $\Pi_{conv}(P, (x))$ enables each party $P_i$ locally computes $[\cdot]x = [c_0 \cdot (x)_1 + c_1 \cdot (x)_1] + c_2 \cdot (x)_2 = [x]0 + [x]1 + [x]2$ (12)

\[ x = c_0 \cdot (x)_1 + c_1 \cdot (x)_1 + c_2 \cdot (x)_2 = [x]0 + [x]1 + [x]2 \]
Here, we only convert three, i.e. \((x)_0, (x)_1, (x)_2\), of the four \((\cdot)^j\)-shares to \([\cdot]\)-shares. Since \textsc{pMPL} supports the privileged party and one or two assistant parties (three shares) to train and the re-computation protocol only needs three shares, this configuration does not affect subsequent operations.

- **Converting** \([\cdot]\)-shares to \((\cdot)^j\)-shares: \(\prod_{j=2}^{\infty}(\mathcal{P}, [x])\) (Protocol 5) enables parties to convert \([\cdot]\)-sharing to \((\cdot)^j\)-sharing. Here, we are supposed to convert three \([\cdot]\)-shares to four \((\cdot)^j\)-shares. Except for the alternate share, each party \(P_i\) locally computes \((x)_j = [x]_{j/c}\). Due to the equation: \(\Phi(3) = a_1 \cdot \Phi(1) + a_2 \cdot \Phi(2)\), we can get the alternate share \((x)_3\) by computing \((x)_3 = a_1 \cdot (x)_1 + a_2 \cdot (x)_2\). We assume that all the parties have already shared a random value \(k\), which is generated in the same way as \(u\) and \(v\) in \(\prod_{i=2}^{\infty}(\mathcal{P})\) (Protocol 4). Then \(P_1\) and \(P_2\) compute \((x) + (k)\) \((j \in \{1, 2\})\) locally, and send them to \(P_0\). Finally, \(P_0\) locally computes the alternate share \((x)_3 = a_1 \cdot (x)_1 + a_1 + a_2 \cdot (x)_2 + (k)_2 - (k)_3\).

**Protocol 5 \(\prod_{j=2}^{\infty}(\mathcal{P}, [x])\)**

**Preprocessing:** Parties pre-shared \((k)\)
**Input:** \([x]\)
**Output:** \((x)\)
1. \(P_1\) locally computes \((x)_j = [x]_{j/c}\).
2. \(P_1\) and \(P_2\) locally compute \((x)_j + (k)\) \((j \in \{1, 2\})\), and send them to \(P_0\).
3. \(P_0\) locally computes \((x)_3 = a_1 \cdot (x)_1 + (k)_1 + a_2 \cdot (x)_2 + (k)_2 - (k)_3\).

**MSB extraction and Bit2A:** The MSB extraction protocol \(\prod_{\text{msbext}}(\mathcal{P}, [x])\) 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 \(\prod_{\text{bit2a}}(\mathcal{P}, [b]^2)\) 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 \textsc{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 \textsc{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, \(P_1\) who holds the secret matrix \(X\) firstly flattens \(X\) into row vector \(X^T\) with the size of \(nd\). Then \(P_1\) constructs a \(3 \times nd\) matrix \(S^T = (X^T, S_1^T, S_2^T)^T\), where \(S_1\) and \(S_2\) are random row vectors with size of \(nd\). Furthermore, \(P_1\) computes shares \((X^T)_k = \Phi(k) \times S^T\) for \(k = \{0, 1, 2, 3\}\). Finally, \(P_1\) converts \((X^T)_k\) to an \(n \times d\) matrix \((X)_k\).

**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 \((X)\) (with the size of \(n \times d\)) and \((Y)\) (with the size of \(d \times n\)), in the matrix addition, each party \(P_i\) locally computes \((Z)_i = (X)_i + (Y)_i\). \(P_0\) additionally computes the alternate shared matrix \((Z)_3 = (X)_3 + (Y)_3\).

![Figure 3: Matrix conversions during matrix sharing](image)

To multiply two shared matrices \((X)\) and \((Y)\), 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 protocol. Here, \(U\) and \(V\) are random matrices, \(H\) has the same dimension as \(X\) and \(H = U \times V\). We assume that all the parties have already shared \((U), (V), (H)\). Each party \(P_i\) firstly computes \((E)_i = (X)_i + (U)_i\) and \((F)_i = (Y)_i + (V)_i\) locally. \(P_0\) additionally computes \((E)_3 = (X)_3 + (U)_3\) and \((F)_3 = (Y)_3 + (V)_3\). Then parties \(P_i\) reveal \(E\) and \(F\), and compute \((Z)_i = (X)_i \times F - E \times (V)_i + (H)_i\) locally. \(P_0\) additionally computes \((Z)_3 = (X)_3 \times F - E \times (V)_3 + (H)_3\).

As for the generation of matrix vector multiplication triplets \((U, V, H)\), the process is similar to \(\prod_{\text{vmult}(\mathcal{P})}\) (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 \((U_0)_i, (U_1)_i, (U_2)_i\). Besides, \(P_0\) additionally holds another three alternate shares \((U_0)_3, (U_1)_3, (U_2)_3\). Then each party \(P_i\) adds these three shared matrices locally to compute \((U)_i = (U_0)_i + (U_1)_i + (U_2)_i\). Additionally, \(P_0\) computes \((U)_3 = (U_0)_3 + (U_1)_3 + (U_2)_3\). For the generation of \((H)\), we generalize the secure computation method proposed by Zhu and Takagi [40] to shared matrices. Firstly, \(P_0\) and \(P_2\) interactively compute \(U_0 \times V_1 + U_1 \times V_0\), \(P_1\) and \(P_2\) interactively compute \(U_0 \times V_2 + U_2 \times V_0\), \(P_1\) and \(P_2\) interactively compute \(U_1 \times V_2 + U_2 \times V_1\). Then each party \(P_i\) locally computes \(H_i = U_i \times V_1 + [U_0 \times V_{i+1} + U_{i+1} \times V_0] + [U_1 \times V_{i+1} + U_{i+1} \times V_1]\). Furthermore, each party \(P_i\) shares \(H_i\) using the matrix sharing protocol. Finally, each party \(P_i\) locally computes \((H)_i = (H_0)_i + (H_1)_i + (H_2)_i\). \(P_0\) additionally computes the alternate shared matrix \((H)_3 = (H_0)_3 + (H_1)_3 + (H_2)_3\).

**Protocol 6 \(\prod_{\text{trunc}}(\mathcal{P}, [z])\)**

**Preprocessing:** Parties pre-shared random values \((r)\) and \((r')\) \((r/2^f)\)
**Input:** \([z]\)
**Output:** The result after truncation \((z')\), where \(z' = z/2^f\)
1. \(P_1\) locally computes \((z - r) = (z)_1 - (r)_1\). \(P_0\) additionally computes \((z - r)_3 = (z)_3 - (r)_3\).
2. \(P_1\) and \(P_2\) send \((z - r)\) and \((z - r)_2\) to \(P_0\) respectively.
3. \(P_0\) locally computes \((z')_0 = (z - r)_2/2^f \times c + (r')_0\) and assistant parties \(P_j\) for \(j \in \{1, 2\}\) holds \((z')_j = (r')_j\). \(P_0\) additionally holds \((z')_3 = (r')_3\).

**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 $2f^2$ 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 $\Pi_{\text{trunc}}(\mathcal{P}, (z))$ proposed in pMPL. At first, we observe that:

$$ z' = \frac{z}{2^{2f}} = \frac{c_0 \cdot (z_0 + c_1 \cdot (z_1) + c_2 \cdot (z_2) + c_3 \cdot (z_3)}{2^{2f}} $$

Additionally, we describe the approximations and computations of these activation functions in machine learning, i.e., ReLU and Sigmod (Protocol 8) to obtain $\{0, 1\}$. Furthermore, each party holds $(x, y)$ and $\Pi_{\text{mul}}(\mathcal{P}, (z), (x), (y))$ (Protocol 3) to compute $(\text{ReLU(x)}).$

### Protocol 7 $\Pi_{\text{trunc}}(\mathcal{P})$

**Input:** 0

**Output:** The truncation pairs $(r, r')$, where $r' = r / 2^{2f}$.

1. Parties generate edabits $(r_0, r_0^2, r_1, r_1^2, \ldots, r_{f-1}, r_{f-1}^2)$, and $(r', r_0', r_1', r_1'^2, \ldots, r_{f-1}', r_{f-1}'^2)$, where $r' = r / 2^{2f}$. After that, each party holds $[\cdot]$-sharing of $r$. Then they interactively execute $\Pi_{\text{mul}}(\mathcal{P}, [r])$ and $\Pi_{\text{mul}}(\mathcal{P}, [r'])$ (Protocol 5) to get $(r)$ and $(r')$.

### Activation Functions

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

- **ReLU:** ReLU function, which is defined as $\text{ReLU}(x) = \max(x, 0)$, can be viewed as $\text{ReLU}(x) = (1 @ b) \cdot x$. The bit $b$ denotes the MSB of $x$, where $b = 1$ if $x < 0$ and $0$ otherwise. $\Pi_{\text{relu}}(\mathcal{P}, (x))$ (Protocol 8) enables parties to compute the shared ReLU function outputs, $(\text{ReLU}(x))$. Firstly, parties interactively execute $\Pi_{\text{mul}}(\mathcal{P}, (x))$ to convert $(x)$ to $[x]$. Then they interactively execute $\Pi_{\text{subex}}(\mathcal{P}, [x])$ on $[x]$ to obtain the shared MSB of $x$, namely $[1 @ b]^2$. Furthermore, each party $P_i$ locally computes $[1 @ b]^2$. Next, parties interactively execute $\Pi_{\text{mul}}(\mathcal{P}, [1 @ b]^2)$ to convert $[1 @ b]^2$ to $(1 @ b)$. After that, parties interactively execute $\Pi_{\text{mul}}(\mathcal{P}, (1 @ b))$ (Protocol 5) to convert $(1 @ b)$ to $(1 @ b)$. At last, parties interactively execute $\Pi_{\text{subex}}(\mathcal{P}, (x))$ (Protocol 3) to compute $(\text{ReLU}(x))$, such that $\text{ReLU}(x) = 0$ if $x < 0$, and $\text{ReLU}(x) = x$ otherwise.

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

$$\text{Sigmod}(x) = \begin{cases} 0, & x \leq \frac{1}{2} \\ x + 1, & \frac{1}{2} < x < \frac{1}{2} \\ 1, & x \geq \frac{1}{2} \end{cases}$$

(14)

This function can be viewed as $\text{Sigmod}(x) = (1 @ b_1) \cdot b_2 \cdot (x + 1 / 2) + (1 @ b_2)$, where $b_1 = 1$ if $x < -1/2$ and $b_2 = 1$ if $x < 1 / 2$. $\Pi_{\text{sig}}(\mathcal{P}, (x))$ is similar to $\Pi_{\text{relu}}(\mathcal{P}, (x))$. We thus do not describe it in detail.

### 4.5 Robustness Design (2PC)

In pMPL, we ensure the robustness throughout 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 $(x + y)$, if $P_2$ drops out, $P_0$ locally computes $(z_0) = (x_0) + (y_0)$, $(z_3) = (x_3) + (y_3)$, and $P_1$ locally computes $(z_1) = (x_1) + (y_1)$.

### Protocol 9 $\Pi_{\text{mul}}(\mathcal{P}, (x), (y))$

**Preprocessing:** Parties pre-shared vector multiplication triplet $(u), (v), (h)$ using $\Pi_{\text{vecmul}}(\mathcal{P})$ (Protocol 4)

**Input:** $(x)$ and $(y)$.

**Output:** $(x + y)$.

1. $P_j$ for $j \in \{0, 1\}$ locally computes $(e_j) = (x_j) + (u_j)$ and $(f_j) = (y_j) + (v_j)$. Besides, $P_0$ computes $(e_3) = (x_3) + (u_3)$ and $(f_3) = (y_3) + (v_3)$.

2. Parties interactively execute $\Pi_{\text{mul}}(\mathcal{P}, (e))$ (Protocol 2) and $\Pi_{\text{mul}}(\mathcal{P}, (f))$ (Protocol 2).

3. $P_j$ for $j \in \{0, 1\}$ locally computes $(z_j) = (x_j) \cdot (f_j) \cdot e + (h_j)$. Besides, $P_0$ computes $(z_3) = (x_3) \cdot (f_3) \cdot (e_3) + (h_3)$.
Protocol 9 shows 2PC secure multiplication protocol $\Pi_{\text{mul}}(P, \langle x, y \rangle)$. Firstly, $P_0$ locally computes $(e_0) = (x_0) + (u_0)$, $(e_3) = (x_3) + (u_3)$ and $(f_3) = (y_0) + (v_0)$. $f_3 = (y_0) + (v_0)$. $P_1$ also locally computes $(e_1) = (x_1) + (u_1)$ and $(f_1) = (y_1) + (v_1)$. Then $P_0$ and $P_1$ interactively execute $\Pi_{\text{rec}}(P, \langle e \rangle)$ (Protocol 2) and $\Pi_{\text{rec}}(P, \langle f \rangle)$ (Protocol 2) to obtain $e$ and $f$ respectively. Finally, $P_0$ computes $(z_0) = (x_0) \cdot f - (e_3) \cdot e + (h_0)$, $(z_3) = (x_3) \cdot f - (e_3) \cdot e + (h_3)$, and $P_1$ computes $(z_1) = (x_1) \cdot f - (e_3) \cdot e + (h_1)$.

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

- **Converting (·)-sharing to \([\cdot]-sharing:** $P_0$ locally computes $[x]_0 = c'_0 \cdot [x]_0$ and $[x]_3 = c'_3 \cdot [x]_3$. Besides, $P_1$ locally computes $[x]_1 = c'_1 \cdot [x]_1$, such that $x = c'_0 \cdot (x_0) + c'_1 \cdot (x_1) + c'_3 \cdot (x_3) = (x_0) + [x]_1 + [x]_3$. Therefore, $P_0$ and $P_1$ convert their (·)-shares to \([\cdot]-shares.

- **Converting \([\cdot]-sharing to (·)-sharing:** $P_0$ locally computes $(x)_0 = [x]_0/c'_0$ and $(x)_3 = [x]_3/c'_3$. Besides, $P_1$ locally computes $(x)_1 = [x]_1/c'_1$.

**Protocol 10** $\Pi_{\text{trunc}}(P, \langle \bar{z} \rangle)$

**Preprocessing:** Parties pre-shared random values $(r)$ and $(r') = (r/2^{2f})$.

**Input:** $(z)$

**Output:** The result after truncation $(z')$, where $z' = z/2^{2f}$

1. $P_1$ for $j \in \{0, 1\}$ locally computes $(z - r)_j = (z)_j - (r)_j$. $P_0$ also computes $(z - r)_3 = (z)_3 - (r)_3$;
2. $P_1$ sends $(z - r)_1$ to $P_0$;
3. $P_0$ locally computes $(z')_0 = (z - r)/2^{2f} \cdot c'_0$ and $(r')_0$ and holds $(z')_3 = (r')_3$. $P_1$ holds $(z')_1 = (r')_1$.

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

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

Protocol 10 shows the 2PC secure truncation protocol $\Pi_{\text{trunc}}(P, \langle z \rangle)$. Firstly, $P_1$ sends $(z - r)$ to $P_0$. Then $P_0$ locally computes $z - r = c'_0 \cdot (z - r)_0 + c'_1 \cdot (z - r)_1 + c'_3 \cdot (z - r)_3$ and $(z - r)/(2^{2f} \cdot c'_0)$ and $(r')_0$. Besides, $P_0$ also holds $(r')_3$ and $P_1$ holds $(r')_1$. 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 round and online communication size in both 3PC (no party drops out) and 2PC (P2 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, $\ell$ denotes the number of bits of a value. $n \times d \times m$ are the sizes for the left and right inputs of matrix-based computations. ReLU and Sigmoid are executed on a single value. $\lambda$ is the security parameter of oblivious transfer in SecureML. Rounds stands for online communication rounds and Comm. stands for online communication size.

| Building block | Framework | IPC | Comm | IPC | Comm |
|----------------|-----------|-----|------|-----|------|
| Matrix addition | pMPL | 0 | 0 | 0 | 0 |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |
| Matrix multiplication | pMPL | 1 | 2d(ad + dm) | 1 | 2d(ad + dm) |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |
| Matrix truncation | pMPL | 1 | 2fm | 1 | 2fm |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |
| Multiplication with truncation | pMPL | 2 | 2d(ad + dm + 2fm) | 2 | 6fm + 1d(ad + dm) |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |
| ReLU | pMPL | log + r/2 | 18n + 4d log(f + 4) | log + r | 8f + 2d log(f + 4) |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |
| Sigmoid | pMPL | log + r/2 | 18n + 4d log(f + 5) | log + r | 4f/2(f + 1) + 6f |
| | SecureML | \(\\) | \(\\) | \(\\) | \(\\) |
| | TF-Encrypted | \(\\) | \(\\) | \(\\) | \(\\) |

#### 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 $\mathbb{Z}_2$. Here, we set $f = 64$, and the least 20 significant bits $f_p$ represent the fractional part, which is the same as the setting of SecureML and TF-Encrypted. Additionally, we set public matrix $\Phi(P)$ as follows:

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

Therefore, according to Equation (6), we can compute $c_0 = 1, c_1 = 2^f - 2, c_2 = 1, c_3 = 2^f - 3, c_4 = 1, c_5 = 2^f - 2$. **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 $n \times E / B$. 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 $119 \times$ faster than SecureML based on HE protocol and around $6 \times$ 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) |
|---------------------|---------------------------|---------------|
|                     | pMPL                      | 100 500 1000  |
| 1,000               | SecureML (HE-based)       | 35.4 79.8 133.3 |
|                     | SecureML (OT-based)       | 0.13 3.78 7.89  |
| 10,000              | SecureML (HE-based)       | 573.9 838.9 1321 |
|                     | SecureML (OT-based)       | 8.68 15.9 31.9  |
| 100,000             | SecureML (HE-based)       | 38.0 78.0 140.28 |
|                     | SecureML (OT-based)       | 8.68 14.0 25.8  |

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:

$$\tilde{w} := \tilde{w} - \frac{\alpha}{B} X_i^T \times (X_i \times \tilde{w} - Y_i)$$

where $X_i$ is a subset of batch size $B$. Besides, $(X_i, Y_i)$ 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 $2.7 \times$ ~ $16.1 \times$ faster and pMPL for 2PC is around $3.8 \times$ ~ $18.6 \times$ 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.

(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:

$$\tilde{w} := \tilde{w} - \frac{\alpha}{B} X_i^T \times (\text{Sigmoid}(X_i \times \tilde{w}) - Y_i)$$

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

| Setting | Dimension (D) | Protocol | Batch Size (B) |
|---------|---------------|----------|----------------|
|         |               | pMPL (3PC) | 128 256 512 1024 |
| 10      | pMPL (2PC)    | 454.45 486.15 2681.58 1666.67 |
| SecureML| 7.899 7.286 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 |
| SecureML| 1428.57 813.01 436.66 202.02 |
| TF-Encrypted| 2612 755 325 281 |
| 1000    | pMPL (2PC)    | 38.05 39.53 17.74 8.87 |
| SecureML| 137.36 58.82 26.39 12.43 |
| TF-Encrypted| 24.53 12.74 6.55 3.30 |
| 1000    | pMPL (3PC)    | 9.68 4.89 4.84 4.73 |
| SecureML| 12.40 12.40 12.40 12.40 |
| TF-Encrypted| 11.58 11.53 11.42 11.15 |
| 100     | pMPL (2PC)    | 4.66 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 |
| 1000    | pMPL (3PC)    | 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 |
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          |
|          | pMPL (3PC)    | 579.45        | 357.47         | 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          |
| LAN     | 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          |
| WAN     | 100           | pMPL (3PC)    | 100.66         | 94.53          | 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           |
|         | 1,000         | pMPL (3PC)    | 9.19           | 5.33           | 2.66           | 1.31           |
|          | pMPL (2PC)    | 10            | 6.89           | 3.54           | 1.80           |
|          | SecureML      | 10            | 6.89           | 3.54           | 1.80           |
|          | TF-Encrypted  | 12.38         | 6.89           | 3.54           | 1.80           |

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

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          |
|          | pMPL (3PC)    | 16.49         | 8.43           | 4.08           | 1.96           |
|          | pMPL (2PC)    | 17.61         | 8.62           | 4.14           | 1.91           |
|          | SecureML      | 29.56         | 18.95          | 11.38          | 6.13           |
|          | TF-Encrypted  | 15.79         | 7.88           | 3.84           | 1.77           |
| LAN     | 100           | pMPL (2PC)    | 16.25         | 8.17           | 3.95           | 1.81           |
|          | SecureML      | 25.39         | 15.78          | 8.63           | 5.02           |
|          | TF-Encrypted  | 12.38         | 6.89           | 3.54           | 1.80           |
| WAN     | 100           | pMPL (2PC)    | 9.19           | 5.33           | 2.66           | 1.31           |
|          | SecureML      | 12.38         | 6.89           | 3.54           | 1.80           |
|          | TF-Encrypted  | 9.19           | 5.33           | 2.66           | 1.31           |
|          | 1,000         | pMPL (2PC)    | 0.15           | 0.12           | 0.10           | 0.07           |
|          | SecureML      | 0.15           | 0.12           | 0.10           | 0.07           |
|          | TF-Encrypted  | 0.15           | 0.12           | 0.10           | 0.07           |
| WAN     | 100           | pMPL (2PC)    | 0.16           | 0.14           | 0.12           | 0.09           |
|          | SecureML      | 0.16           | 0.14           | 0.12           | 0.09           |
|          | TF-Encrypted  | 0.16           | 0.14           | 0.12           | 0.09           |
|          | 1,000         | pMPL (2PC)    | 0.14           | 0.12           | 0.09           | 0.06           |
|          | SecureML      | 0.15           | 0.13           | 0.11           | 0.08           |
|          | TF-Encrypted  | 0.14           | 0.12           | 0.09           | 0.06           |

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.

6 DISCUSSION

6.1 pMPL with More Assistant Parties

Our proposed pMPL can be extended to support more assistant parties by setting public matrix $\Phi(P)$. In order to support more assistant parties, we can increase the number of columns of the public matrix $\Phi(P)$, i.e. expand the dimension of each public vector $\Phi(i)$. For instance, for a set of parties $P = \{P_0, P_1, P_2, P_3, P_4\}$ and an access structure $\mathcal{F} = \{B_0, B_1, B_2, B_3, B_4\} = \{\{P_0, P_1, P_2, P_3, P_4\}, \{P_0, P_1, P_2, P_3\}, \{P_0, P_1, P_2, P_3\}, \{P_0, P_1, P_2, P_3\}\}$, 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 $\Phi(P)$ with the size of $6 \times 5$ should satisfy the following four restrictions:

1. $(1, 0, 0, 0, 0)$ 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.

2. 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, $\Phi(5) = \sum_{j=1}^{4} a_j \cdot \Phi(j)$, 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.

3. 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 $\{\Phi(1), \Phi(2), \Phi(3), \Phi(4), \Phi(5)\}$ and their subsets. The values of public matrix $\Phi(P)$ and reconstruction coefficients should be elements of the ring $\mathbb{Z}_{2^2}$.
For example, a public matrix \( \Phi(\mathcal{P}) \) that satisfies the above restrictions is:

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

Note that we can hereby tolerate more assistant parties (\( \leq 3 \)) 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 participate. 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 quits the training, the training process cannot be restarted.

In our proposed \( \Phi^{MPL} \), 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 collide 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, \( \Phi^{MPL} \) can tolerate that one of assistant parties drops out.

### 6.3 Complex Models in MPL Frameworks

\( \Phi^{MPL} \) 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 \( \Phi^{MPL} \), we conduct several experiments on more complex datasets, including Fashion-MNIST and SVHN. We compare the training accuracy of machine learning models trained with \( \Phi^{MPL} \) 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 \( \Phi^{MPL} \) 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, \( \Phi^{MPL} \) now has the potential to support CNN because \( \Phi^{MPL} \) 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 \( \Phi^{MPL} \) 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 \( \Phi^{MPL} \). In \( \Phi^{MPL} \), 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 \( \Phi^{MPL} \) 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 \( \Phi^{MPL} \), an MPL framework based on the vector space secret sharing. To the best of our knowledge, \( \Phi^{MPL} \) is the first academic work to support a privileged party in an MPL framework. \( \Phi^{MPL} \) guarantees that even if two assistant parties collude with each other, only the privileged party can obtain the final result. Furthermore, \( \Phi^{MPL} \) tolerates one of the two assistant parties dropping out during training. That is, \( \Phi^{MPL} \) protects the interests of the privileged party while improving the robustness of the framework. Finally, the experimental results show that the performance of \( \Phi^{MPL} \) is promising when we compare it with state-of-the-art MPL frameworks. Especially, for the linear regression, \( \Phi^{MPL} \) is 16x faster than TF-encrypted and 5x for logistic regression in the LAN setting. In the WAN setting, although \( \Phi^{MPL} \) is slower than both SecureML and TF-encrypted, the performance is still promising. Because \( \Phi^{MPL} \) 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.
We show the shares held by each party \( P \) during the execution of secure multiplication protocol \( \Pi_{mul}(P, (x), (y)) \) (Protocol 3) in Table 7. More specifically, for the first line, each party \( P_i \) holds \((u_i), (e_i), (h_i)\) by performing \( \Pi_{vmtgen}(P) \) (Protocol 4) during the offline phase. \( P_3 \) additionally holds \((u_3), (s_3), (h_3)\). 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 \( \Pi_{mul}(P, (x), (y)) \) (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 \( \Pi_{vmtgen}(P) \) (Protocol 4) in Table 8. More specifically, the three steps of generating \((u_i), (e_i), (h_i)\) is corresponding to the first three lines of Table 8. For the four steps of generating \((h_i)\), 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 \( S \) to denote an ideal-world static adversary (simulator) for a real-world adversary. \( S \) 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 for corrupt \( P_1 \) and \( P_2 \) as shown in Figure 5 and Figure 6 respectively.

Figure 4: Functionality \( F_{shr} \)

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.

Figure 5: Simulator \( S_{shr}^{P_0} \)

Figure 6: Simulator \( S_{shr}^{P_1, P_2} \)

We denote \( \text{view}_{P_0}^{shr} \) and \( \text{view}_{P_1, P_2}^{shr} \) as the views of \( P_0 \) and \( P_1, P_2 \) respectively. We note that \( P_0 \)'s view and \( S_{shr}^{P_0} \)'s output are identical, the probability distribution of \( P_1 \) and \( P_2 \)'s views and \( S_{shr}^{P_1, P_2} \)'s output are identical. Therefore we have the following equations:

\[
S_{shr}^{P_0}(x, (x_0), (x_3)) \equiv \text{view}_{P_0}^{shr}(x, (x)_k, k \in \{0, 1, 2, 3\})
\]

\[
S_{shr}^{P_1, P_2}(x, (x)_2) \equiv \text{view}_{P_1, P_2}^{shr}(x, (x)_k, k \in \{0, 1, 2, 3\})
\]

Reconstruction Protocol: The ideal functionality \( F_{rec} \) realising reconstruction protocol \( \Pi_{rec}(P, (x)) \) (Protocol 2) is presented in Figure 7. Here, we only consider the case of no party drops out.

Theorem 2. Reconstruction protocol \( \Pi_{rec}(P, (x)) \) (Protocol 2) securely realises the functionality \( F_{rec} \) (Figure 7) in the presence of static semi-honest adversary.
Table 7: Shares held by each party during the execution of $\prod_{\text{mul}}(P, (x), (y))$ (Protocol 3). For each line, the shares held by each party $P_i$ correspond to each step in $\prod_{\text{mul}}(P, (x), (y))$ (Protocol 3).

| Step | Privileged party $P_0$ | Assistant party $P_1$ | Assistant party $P_2$ |
|------|------------------------|------------------------|------------------------|
| Pre-generating | $(u)_0, (u)_3, (\phi)_0, (\phi)_3, (h)_0, (h)_3$ | $(u)_1, (v)_1, (h)_1$ | $(u)_2, (v)_2, (h)_2$ |
| Inputting | $(x)_0, (x)_3, (y)_0, (y)_3$ | $(x)_1, (y)_1$ | $(x)_2, (y)_2$ |
| Locally computing | $(e)_0 = (x)_0 + (u)_0$ | $(e)_1 = (x)_1 + (u)_1$ | $(e)_2 = (x)_2 + (u)_2$ |
| Communicating | $\prod_{\text{mul}}(P, (x))$ and $\prod_{\text{mul}}(P, (y))$ | $\prod_{\text{mul}}(P, (e))$ and $\prod_{\text{mul}}(P, (f))$ |
| Locally computing | $(z)_0 = (x)_0 \cdot f - (v)_0 \cdot e + (h)_0$ | $(z)_1 = (x)_1 \cdot f - (v)_1 \cdot e + (h)_1$ | $(z)_2 = (x)_2 \cdot f - (v)_2 \cdot e + (h)_2$ |

Table 8: Shares held by each party during the execution of $\prod_{\text{vmtgen}}(P)$ (Protocol 4). For each line, the shares held by each party correspond to each step in $\prod_{\text{vmtgen}}(P)$ (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 $\prod_{\text{vmtgen}}(P, u_1)$ and $\prod_{\text{vmtgen}}(P, v_1)$ | $(u)_0, (u)_1, (u)_2, (u)_3$ | $(u)_1, (u)_2, (u)_3$ | $(u)_2, (u)_3, (u)_4, (u)_5$ |
| Locally computing | $(o)_0 = (u)_0 + (u)_1 + (u)_2$ | $(o)_1 = (u)_1 + (u)_2 + (u)_3$ | $(o)_2 = (u)_2 + (u)_3 + (u)_4$ |
| Secure computing | $[u_0 + v_0 + u_1]_1$ | $[u_1 + v_1 + u_2]_1$ | $[u_2 + v_2 + u_3]_1$ |
| Executing $\prod_{\text{vmtgen}}(P, h_1)$ | $(h)_0, (h)_1, (h)_2, (h)_3$ | $(h)_1, (h)_2, (h)_3$ | $(h)_2, (h)_3, (h)_4, (h)_5$ |
| Locally computing | $(b)_0 = (h)_0 + (h)_1 + (h)_2$ | $(b)_1 = (h)_1 + (h)_2 + (h)_3$ | $(b)_2 = (h)_2 + (h)_3 + (h)_4$ |

Functionality $F_{\text{rec}}$

Input:
- $P_0$ inputs $(x)_0$;
- $P_1$ inputs $(x)_1$;
- $P_2$ inputs $(x)_2$.

Output:
- $P_0$, $P_1$ and $P_2$ all output $x$.

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 $\text{view}_{P_0}$ and $\text{view}_{P_1, P_2}$, as the views of $P_0$ and $P_1$, $P_2$ respectively. We note that the probability distribution of $P_0$’s view and $S_{\text{rec}}^{P_0}$’s output are identical, the probability distribution of $P_1$ and $P_2$’s views and $S_{\text{rec}}^{P_1, P_2}$’s output are identical. Therefore we have the following equations:

$$S_{\text{rec}}^{P_0}(x)_0, x) \equiv \text{view}_{P_0}^{\text{rec}}((x)_0, (x)_1, (x)_2, x)$$
$$S_{\text{rec}}^{P_1, P_2}(x)_1, (x)_2, x) \equiv \text{view}_{P_1, P_2}^{\text{rec}}((x)_0, (x)_1, (x)_2, x)$$

**Simulator $S_{\text{rec}}^{P_0, P_1}$**

1. $S_{\text{rec}}^{P_0}$ receives $(x)_0$ and $c_0, c_1, c_2$ from $P_0$.
2. $S_{\text{rec}}^{P_0}$ selects two random values $(x)_1, (x)_2$.
3. $S_{\text{rec}}^{P_0}$ computes
   $$x = c_0 \cdot (x)_0 + c_1 \cdot (x)_1 + c_2 \cdot (x)_2$$
4. $S_{\text{rec}}^{P_0}$ outputs $((x)_0, (x)_1, (x)_2, x)$.

**Multiplication Protocol** The ideal functionality $F_{\text{mul}}$ realising multiplication protocol $\prod_{\text{mul}}(P, (x), (y))$ (Protocol 3) is presented in Figure 10.
respectively. We note that the probability distribution of $p_{MPL}$ for $P_0$, $P_1$, and $P_2$ are:

$$
\begin{align*}
P_0 &= \langle x \rangle_0, \langle y \rangle_0, \langle x \rangle_3, \langle y \rangle_3 \\
P_1 &= \langle x \rangle_1, \langle y \rangle_1 \\
P_2 &= \langle x \rangle_2
\end{align*}
$$

and $S_{mul}^{P_0}$'s output are identical, $P_1$ and $P_2$'s view and $S_{mul}^{P_1, P_2}$'s output are identical. Therefore we have the following equations:

$$
\begin{align*}
S_{mul}^{P_0}(\langle x \rangle_0, \langle y \rangle_0, \langle x \rangle_3, \langle y \rangle_3, \langle z \rangle_0, \langle z \rangle_3) &= \equiv \\
\text{view}_{P_0}^{mul}(\langle x \rangle_0, \langle y \rangle_0, \langle z \rangle_0, \langle z \rangle_3) &= \equiv \\
S_{mul}^{P_1, P_2}(\langle x \rangle_1, \langle y \rangle_1, \langle z \rangle_1, \langle z \rangle_2) &= \equiv \\
\text{view}_{P_1, P_2}^{mul}(\langle x \rangle_1, \langle y \rangle_1, \langle z \rangle_1, \langle z \rangle_2) &= \equiv
\end{align*}
$$

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

**Theorem 4.** Sharing conversion protocol $\Pi_{a2v}(P, [x])$ (Protocol 5) securely realizes the functionality $F_{a2v}$ (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.
Truncation Protocol: The ideal functionality $F^\text{trunc}$ realizing truncation protocol $\Pi^\text{trunc}(\mathcal{P}, (\zeta))$ (Protocol 6) is presented in Figure 16.

$F^\text{trunc}$

Input:
- $P_0$ inputs $\langle \zeta \rangle_0$;
- $P_1$ inputs $\langle \zeta \rangle_1$;
- $P_2$ inputs $\langle \zeta \rangle_2$.

Output:
- $P_0$ outputs $\langle \zeta' \rangle_0$ and $\langle \zeta' \rangle_3$;
- $P_1$ outputs $\langle \zeta' \rangle_1$;
- $P_2$ outputs $\langle \zeta' \rangle_2$, where $\zeta' = \zeta / 2^f$.

Theorem 5. Truncation protocol $\Pi^\text{trunc}(\mathcal{P}, (\zeta))$ (Protocol 6) securely realizes the functionality $F^\text{trunc}$ (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.

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

$S^\text{a2v}_{P_0}(\{x_0, x_1, x_2, x_3\}, (\zeta)_0, (\zeta)_1, (\zeta)_2) \equiv \text{view}_{\mathcal{P}_0}^{a2v}(\{x_i\}, (\zeta)_k, k \in \{0, 1, 2, 3\})$

$S^\text{a2v}_{P_1, P_2}(\{x_0, x_1, x_2, x_3\}, (\zeta)_1, (\zeta)_2) \equiv \text{view}_{\mathcal{P}_1, \mathcal{P}_2}^{a2v}(\{x_i\}, (\zeta)_k, k \in \{0, 1, 2, 3\})$

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 $28 \times 28$ 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 $32 \times 32$ 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 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 9: Brief description of datasets used in pMPL.

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

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        | Accuracy          |
|----------------|----------------|-------------------|
|                |                | pMPL   | Plaintext |
| Linear regression | MNIST          | 85.77% | 85.80%    |
| Logistic regression | Fashion-MNIST | 80.69% | 80.80%    |
| BP neural networks | MNIST          | 91.05% | 91.38%    |
|                | Fashion-MNIST  | 83.99% | 84.01%    |
|                | SVHN           | 73.31% | 73.35%    |