Robust and Secure Federated Learning with Low-Cost Zero-Knowledge Proof

Yizheng Zhu
National University of Singapore

Yuncheng Wu
National University of Singapore

Zhaojing Luo
National University of Singapore

Beng Chin Ooi
National University of Singapore

Xiaokui Xiao
National University of Singapore

Date: 1 September 2023

Abstract

Federated Learning (FL) enables multiple clients to collaboratively train a machine learning (ML) model under the supervision of a central server while ensuring the confidentiality of their raw data. However, existing studies have unveiled two main risks: (i) the potential for the server to infer sensitive information from the client's uploaded updates (i.e., model gradients), compromising client input privacy, and (ii) the risk of malicious clients uploading malformed updates to poison the FL model, compromising input integrity. Recent works utilize secure aggregation with zero-knowledge proofs (ZKP) to ensure input privacy and input integrity simultaneously in FL. Nevertheless, they suffer from extremely low efficiency and, thus, are impractical for real deployment. In this paper, we propose a novel solution RiseFL, which is robust, secure, and highly efficient to guarantee input privacy and integrity. Firstly, we devise a probabilistic integrity check method, significantly reducing the cost of ZKP generation and verification. Secondly, we design a hybrid commitment scheme to satisfy Byzantine robustness with improved performance. Thirdly, we theoretically prove the security guarantee of the proposed solution. Extensive experiments on synthetic and real-world datasets suggest that our solution is effective and is highly efficient in both client computation and communication. For instance, RiseFL is up to 53x and 164x faster than two state-of-the-art baselines RoFL and EIFFeL for the client computation.

1 Introduction

Federated Learning (FL) [20,31,32,35,39,53] is an emerging paradigm that enables multiple data owners (i.e., clients) to collaboratively train a machine learning (ML) model without sharing their private data with each other. Typically, there is a centralized server that coordinates the FL training process as follows. The server first initializes the model parameter and broadcasts it to all clients. Then, in each iteration, each client computes a local update (i.e., model gradients) on its own data and uploads it to the server. The server aggregates all clients' updates to generate a global update and sends it back to the clients for iterative training [35].

Despite the fact that FL could facilitate data collaboration among multiple clients, two main risks remain, as illustrated in Figure 1. The first is the client's *input privacy*. Even without disclosing the client's raw data to the server, recent studies [36,38,56,58] have shown that the server can recover the client's sensitive data through the uploaded update with a high probability. The second is the client's *input integrity*. In FL, there may exist a set of malicious clients that aim to poison the FL model via Byzantine attacks, such as contaminating the training process with malformed updates to degrade the model accuracy [4,18,21,25], imposing backdoors so that the FL model is susceptible to specific types of inputs [2,12,40,51], and so on.

A number of solutions [1,3,5,7,11,28,34,41,49,52,54,55,57] have been proposed to protect input privacy and ensure input integrity in FL. On the one hand, instead of uploading the plaintext local updates to the server, the clients can utilize secure aggregation techniques [3,7,28,57], such as secret sharing [29,46] and homomorphic encryption [16,22], to mask or encrypt the local updates so that the server can aggregate the clients' updates correctly without knowing each update. In this way, the client's input privacy is preserved. However, these solutions do not ensure input integrity because it is difficult to distinguish a malicious encrypted update from benign ones. On the other hand, [1,5,11,34,41,52,54,55] present various Byzantine-robust aggregation algorithms, allowing the server to identify malformed updates and eliminate them from being aggregated into the global update. Nevertheless, these algorithms require the clients to send plaintext updates to the server for the integrity check, which compromises the client's input privacy.

In order to ensure input integrity while satisfying input privacy, [9,45] use secure aggregation to protect each client's update and allow the server to check the encrypted update's integrity using zero-knowledge proof (ZKP) protocols. The

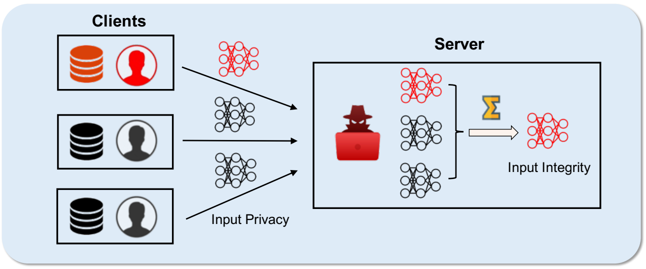


Figure 1: The input privacy and input integrity risks in FL.

general idea is to let each client compute a commitment of its local update and generate a proof that the update satisfies a publicly-known predicate, for example, the L_2-norm is within a specific range; then, the server can verify the correctness of the proofs based on the commitments without the need of knowing the plaintext values and securely aggregate the valid updates. However, these solutions suffer from extremely low efficiency in proof generation and verification, making them impractical for real deployments.

To introduce a practical FL system which ensures both input privacy and input integrity, we propose a robust and secure federated learning approach, called RiseFL, with high efficiency. In this paper, we focus on the L_2-norm integrity check, i.e., the L_2-norm of a client's local update is less than a threshold, which is widely adopted in existing works [9, 13, 45, 50]. Our key observation of the low-efficiency in [9, 45] is that their proof generation and verification costs are linearly dependent on the number of parameters d in the FL model. Therefore, we aim to reduce the proof cost, which particularly makes sense in the FL scenario because the clients often have limited computation and communication resources. Our approach has the following novelties. First, we propose a probabilistic L_2-norm check method that decreases the cryptographic operation cost of proof generation and verification from $O(d)$ to $O(d/\log d)$. The intuitive idea is to sample a set of public vectors and let each client generate proofs with respect to the inner product between its update and each public vector, instead of checking the L_2-norm of the update directly. This allows us to reduce the proof generation and verification time significantly. Second, we devise a hybrid commitment scheme based on Pedersen commitment [42] and verifiable Shamir secret sharing commitment [19, 46], ensuring Byzantine-robustness while achieving further performance improvement on the client computation. Third, although we consider the L_2-norm check, the proposed approach can be easily extended to various Byzantine-robust integrity checks [5, 11, 48, 54] based on different L_2-norm variants, such as cosine similarity, sphere defense [48], and so on.

In summary, we make the following contributions.

- We propose a novel and highly efficient federated learning solution RiseFL that simultaneously ensures each client's input privacy and input integrity.
- We present a probabilistic L_2-norm integrity check method and a hybrid commitment scheme, which significantly reduces the ZKP generation and verification costs.
- We provide a formal security analysis of RiseFL and theoretically compare its computational and communication costs with state-of-the-art solutions.
- We implement RiseFL and evaluate its performance with a set of micro-benchmark experiments as well as FL tasks on two real-world datasets. The results demonstrate that RiseFL is effective in detecting malformed updates and is up to 53x and 164x faster than RoFL [9] and EIFFeL [45] for the client computation, respectively.

The rest of the paper is organized as follows. Section 2 introduces preliminaries and Section 3 presents an overview of our solution. We detail the system design in Section 4 and analyze the security and cost in Section 5. The evaluation is provided in Section 6. We review the related works in Section 7 and conclude the paper in Section 8.

2 Preliminaries

We first describe the notations used in this paper. Let $\mathbb{G}$ denote a cyclic group with prime order p, where the discrete logarithm problem [44] is hard. Let $\mathbb{Z}_p$ denote the set of integers modulo the prime p. We use x, $\mathbf{x}$, and $\mathbf{X}$ to denote a scalar, a vector, and a matrix, respectively. We use $\mathsf{Enc}_\mathsf{K}(x)$ to denote an encrypted value of x under an encryption key K, and $\mathsf{Dec}_\mathsf{K}(y)$ to denote a decrypted value of y under the same key K. Since the datasets used in machine learning (ML) are often in the floating-point representation, we use fixed-point integer representation to encode floating-point values.

2.1 Cryptographic Building Blocks

Pedersen Commitment. A commitment scheme is a cryptographic primitive that allows one to commit a chosen value without revealing the value to others while still allowing the ability to disclose it later [24]. Commitment schemes are widely used in various zero-knowledge proofs. In this paper, we use the Pedersen commitment [42] for a party to commit its secret values. Given the independent group elements (g, h), the Pedersen commitment encrypts a value $x \in \mathbb{Z}_p$ to $C(x, r) = g^x h^r$, where $r \in \mathbb{Z}_p$ is a random number. An important property of the Pedersen commitment is that it is additively homomorphic. Given two values x_1, x_2 and two random numbers r_1, r_2, the commitment follows: $C(x_1, r_1) \cdot C(x_2, r_2) = C(x_1 + x_2, r_1 + r_2)$.

Verifiable Shamir's Secret Sharing Scheme. Shamir's t-out-of-n secret sharing (SSS) scheme [46] allows a party to distribute a secret among a group of n parties via shares so that the secret can be reconstructed given any t shares but cannot be revealed given less than t shares. The SSS scheme is verifiable (aka. VSSS) if auxiliary information is provided to verify the validity of the secret shares. We use the VSSS scheme [19, 46] to share a number $r \in \mathbb{Z}_p$. Specifically, the scheme consists of three algorithms SS.Share, SS.Verify, and SS.Recover.

- $((1,r_1),\ldots,(n,r_n),\Psi) \leftarrow \mathsf{SS.Share}(r,n,t,g)$. Given a secret $r \in \mathbb{Z}_p$, $g \in \mathbb{G}$, and $0 < t \le n$, this algorithm outputs a set of n shares (i,r_i) for $i \in [n]$ and a check string Ψ as the auxiliary information to verify the shares. Specifically, it generates a random polynomial f in $\mathbb{Z}_p$ of degree at most $t-1$ whose constant term is r. We set $r_i = f(i)$ and $\Psi = (g^r, g^{f_1}, \ldots, g^{f_{t-1}})$ where f_i is the i-th coefficient.
- $r \leftarrow \mathsf{SS.Recover}(\{(i,r_i)\} : i \in A)$. For any subset $A \subset [n]$ with size at least t, this algorithm recovers the secret r.
- True/False $\leftarrow \mathsf{SS.Verify}(\Psi,i,r_i,n,t,g)$. Given a share (i,r_i) and the check string Ψ, it verifies the validity of this share such that it outputs True if (i,r_i) was indeed generated by $\mathsf{SS.Share}(r,n,t,g)$ and False otherwise.

This scheme is additively homomorphic in both the shares and check string. If $((1,r_1),\ldots,(n,r_n),\Psi_r) \leftarrow \mathsf{SS.Share}(r,n,t,g)$ and $((1,s_1),\ldots,(n,s_n),\Psi_s) \leftarrow \mathsf{SS.Share}(s,n,t,g)$, then:

- $r+s \leftarrow \mathsf{SS.Recover}(\{(i,r_i+s_i)\} : i \in A)$ for any subset $A \subset [n]$ with size at least t,
- True $\leftarrow \mathsf{SS.Verify}(\Psi_r \cdot \Psi_s, i, r_i+s_i, n,t,g)$.

Zero-Knowledge Proofs. A zero-knowledge proof (ZKP) allows a prover to prove to a verifier that a given statement is true, such as a value is within a range, without disclosing any additional information to the verifier [6]. We utilize two ZKP protocols based on Pedersen commitment as building blocks.

The first ZKP protocol is the Σ-protocol [10] for proof of square and proof of relation. For proof of square $((x,r_1,r_2),(y_1,y_2))$, denote $g,h \in \mathbb{G}$ the independent group elements, and let $y_1 = g^x h^{r_1}$ and $y_2 = g^{x^2} h^{r_2}$ be the commitments, where $x,r_1,r_2 \in \mathbb{Z}_p$ are the secrets. To handle the square in power, we rewrite $y_2 = y_1^x h^{r_2 - r_1 x}$. The function $\mathsf{GenPrfSq}()$[1] generates a proof π that (y_1,y_2) is of the form $(g^x h^{r_1}, g^{x^2} h^{r_2})$ for $x,r_1,r_2 \in \mathbb{Z}_p$. Accordingly, the function $\mathsf{VerPrfSq}()$ verifies this proof based on (y_1,y_2). For proof of a relation $((r,v,s),(z,e,o))$, denote $g,q,h \in \mathbb{G}$ the independent group elements, and we let $z = g^r$, $e = g^v h^r$, $o = g^v q^s$ be the commitments, where $r,v,s \in \mathbb{Z}_p$ are the secrets. Then, the function $\mathsf{GenPrfWf}()$ generates a proof π that (z,e,o) is of the form $(g^r, g^v h^r, g^v q^s)$ given x, r and s, and the function $\mathsf{VerPrfWf}()$ verifies the proof π according to (z,e,o).

[1] We detail the $\mathsf{GenPrfSq}()$, $\mathsf{VerPrfSq}()$, $\mathsf{GenPrfWf}()$, $\mathsf{VerPrfWf}()$ functions with batch forms in Algorithms 5-8 in Appendix A.1, respectively.

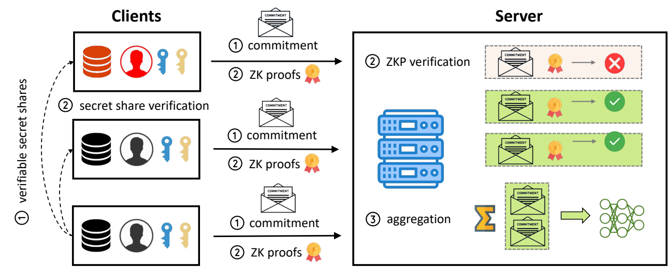


Figure 2: An overview of the proposed RiseFL system.

The second ZKP protocol used is the Bulletproofs protocol [8] for checking the bound of $\mathbf{x} = (x_1,\ldots,x_k)$ in the vector of Pedersen commitments $\mathbf{y} = (g^{x_1}h^{r_1},\ldots,g^{x_k}h^{r_k})$. To generate and verify a proof that $x_j \in [0,2^b)$ for every $j = \{1,\ldots,k\}$, we express x_j in binary: $x_j = \sum_{i=0}^{b-1} x_{ji} 2^i$, each $x_{ji} \in \{0,1\}$. Then, we use group elements $\mathbf{f} \in \mathbb{G}^{2bk}$ to commit x_{ji} and $x_{ji}-1$ for each $j = \{1,\ldots,k\}$ and each $i \in \{0,\cdots,b-1\}$, and use the algorithm in [8] to prove and check that $x_{ji}(x_{ji}-1) = 0$. We denote $\mathsf{GenPrfBd}(g,h,\mathbf{f},b,\mathbf{y},\mathbf{x},\mathbf{r})$ and $\mathsf{VerPrfBd}(g,h,\mathbf{f},b,\mathbf{y},\pi)$ the proof generation and verification functions for a statement that $x_j \in [0,2^b)$ for every j, and refer the interested readers to [8] for more details.

3 System Overview

In this section, we present the system model and threat model, and give an overview of the proposed RiseFL system to ensure input privacy and input integrity.

3.1 System Model

There are n clients $\{\mathcal{C}_1,\cdots,\mathcal{C}_n\}$ and a centralized server in the system. Each client $\mathcal{C}_i (i \in [1,n])$ holds a private dataset $\mathcal{D}_i$ to participate in the federated learning (FL) process for training an FL model $\mathcal{M}$. Let d be the number of parameters in $\mathcal{M}$. In each iteration, the training process consists of three steps. Firstly, the server broadcasts the current model parameters to all the clients. Secondly, each client $\mathcal{C}_i$ locally computes a model update (i.e., gradients) $\mathbf{u}_i$ given the model parameters and its dataset $\mathcal{D}_i$, and submits $\mathbf{u}_i$ to the server. Thirdly, the server aggregates the clients' gradients to a global update $\mathcal{U} = \sum_{i \in [1,n]} \mathbf{u}_i$ and updates the model parameters of $\mathcal{M}$ for the next round of training until convergence.

3.2 Threat Model

We consider a malicious threat model in two aspects. First, regarding input privacy, we consider a malicious server (i.e., the adversary) that can deviate arbitrarily from the specified protocol to infer each client's uploaded model update. Also, the server may collude with some of the malicious clients to compromise the honest clients' privacy. Similar to [45], we do not consider the scenario that the server is malicious against the input integrity because its primary goal is to ensure the well-formedness of each client's uploaded update. Second, regarding input integrity, we assume there are at most m malicious clients in the system, where $m < n/2$. The malicious clients can also deviate from the specified protocol arbitrarily, such as sending malformed updates to the server to poison the aggregation of the global update, or intentionally marking an honest client as malicious to interfere with the server's decision on the list of malicious clients.

3.3 Problem Formulation

We aim to ensure both input privacy (for the clients) and input integrity (for the server) under the threat model described in Section 3.2. Our problem is similar to the secure aggregation with verified inputs (SAVI) problem in [45], but relaxing the input integrity check for efficiency. Definition 1 formulates our problem, namely (D,F)-relaxed SAVI.

Definition 1. *Given a security parameter* κ*, a function* $D : \mathbb{R}^d \to \mathbb{R}$ *satisfying that* $\boldsymbol{u}$ *is malicious if and only if* $D(\boldsymbol{u}) > 1$*, a function* $F : (1,+\infty) \to [0,1]$*, a set of inputs* $\{\boldsymbol{u}_1,\cdots,\boldsymbol{u}_n\}$ *from clients* $\mathcal{C} = \{\mathcal{C}_1,\ldots,\mathcal{C}_n\}$ *respectively, and a list of honest clients* $\mathcal{C}_H$*, a protocol* Π *is a* (D,F)*-relaxed SAVI protocol for* $\mathcal{C}_H$ *if:*

- *Input Privacy. The protocol* Π *realizes the ideal functionality* $\mathcal{F}$ *such that for an adversary* $\mathcal{A}$ *that consists of the malicious server and the malicious clients* $\mathcal{C}_M = \mathcal{C} \setminus \mathcal{C}_H$ *attacking the real interaction, there exist a simulator* $\mathcal{S}$ *attacking the ideal interaction, and*

  $$|\Pr[\mathrm{Real}_{\Pi,\mathcal{A}}(\{\boldsymbol{u}_{\mathcal{C}_H}\}) = 1] - \Pr[\mathrm{Ideal}_{\mathcal{F},\mathcal{S}}(\mathcal{U}_H) = 1]| \le \mathrm{negl}(\kappa),$$

  where $\mathcal{U}_H = \sum_{\mathcal{C}_i \in \mathcal{C}_H} \boldsymbol{u}_i$*.*

- *Input Integrity. The protocol* Π *outputs* $\sum_{\mathcal{C}_i \in \mathcal{C}_{\mathrm{Valid}}} \boldsymbol{u}_i$ *with probability at least* $1 - \mathrm{negl}(\kappa)$*, where* $\mathcal{C}_H \subseteq \mathcal{C}_{\mathrm{Valid}}$*. For any malformed input* $\boldsymbol{u}_j$ *from a malicious client* $\mathcal{C}_j$*, the probability that it passes the integrity check satisfies:*

  $$\Pr[\mathcal{C}_j \in \mathcal{C}_{\mathrm{Valid}}] \le F(D(\boldsymbol{u}_j)).$$

For input privacy in Definition 1, it achieves the same privacy level as that in EIFFeL [45], ensuring that the server can only learn the aggregation of honest clients' updates. For input integrity, Definition 1 relaxes the integrity check by introducing a malicious pass rate function F. F is a function that maps the degree of maliciousness of an input to the pass rate of the input. The degree of maliciousness is measured by the function D. For instance, with an L_2-norm bound B, a natural choice is $D(\mathbf{u}) = \|\mathbf{u}\|_2/B$. Intuitively, the higher the degree of maliciousness, the lower the pass rate. So F is usually decreasing. Our system also satisfies $\limsup_{x\to+\infty} F(x) \le \mathrm{negl}(\kappa)$, which means that any malicious client's malformed update whose degree of maliciousness passes a threshold can be detected with an overwhelming probability, ensuring robustness of the system. When $F \equiv \mathrm{negl}(\kappa)$, a protocol that satisfies (D,F)-relaxed SAVI will also satisfy SAVI.

Algorithm 1 Probabilistic L_2-norm bound check

Input: $\mathbf{u} \in \mathbb{R}^d$, B, k, ε
Output: "Pass" or "Fail"
Sample $\mathbf{a}_1,\ldots,\mathbf{a}_k \in \mathbb{R}^d$ i.i.d. from $\mathcal{N}(\mathbf{0},\mathbf{I}_d)$
Compute $\gamma_{k,\varepsilon}$ which satisfies that $\Pr_{t\sim\chi_k^2}[t < \gamma_{k,\varepsilon}] = 1-\varepsilon$.
if $\sum_{i=1}^k \langle \mathbf{a}_i, \mathbf{u}\rangle^2 \le B^2\gamma_{k,\varepsilon}$ **then**
  Return "Pass"
else
  Return "Fail"
end if

3.4 Solution Overview

To solve the problem in Definition 1, we propose an efficient, robust, and secure federated learning system RiseFL. It tolerates $m < n/2$ malicious clients for input integrity, which means the server can securely aggregate the clients' inputs as long as a majority of the clients are honest. Figure 2 gives an overview of RiseFL, which is composed of a system initialization stage and three iterative rounds: commitment generation, proof generation and verification, and aggregation. In the initialization stage, all the parties agree on some hyper-parameters, such as the number of clients n, the maximum number of malicious clients m, the security parameters (e.g., key size), and so on.

In each iteration of the FL training process, each client $\mathcal{C}_i (i \in [n])$ commits its model update $\mathbf{u}_i$ using the hybrid commitment scheme based on Pedersen commitment and verifiable Shamir's secret sharing (VSSS) in Section 4.2, and sends the commitment to the server and the secret shares to the corresponding clients. In the proof generation and verification round, there are two steps. In the first step, each client verifies the authenticity of other clients' secret shares. For secret shares that are verified to be invalid, the client marks the respective clients as malicious. With the marks from all clients, the server can identify a subset of malicious clients. In the second step, the server uses a probabilistic integrity check method presented in Section 4.3 to check each client's update $\mathbf{u}_i$. Next, the server filters out the malicious client list $\mathcal{C}^*$ and broadcasts it to all the clients. In the aggregation round, each client aggregates the secret shares from clients $\mathcal{C}_j (j \notin \mathcal{C}^*)$ and sends the result to the server. The server can reconstruct the sum of secret shares and securely aggregate the updates $\mathbf{u}_j (j \notin \mathcal{C}^*)$ based on the Pedersen commitments.

1. The client and server agree on independent $g, q \in \mathbb{G}$, $\mathbf{w} \in \mathbb{G}^d$, $\mathbf{f} \in \mathbb{G}^{2kb_{\max}}$, the bound B_0 of sum of squares of inner products, the maximum number of bits $b_{\max}$ of B_0, the number of bits of each inner product $b_{\text{ip}} < b_{\max}$.
2. The client sends $z_i = g^{r_i} \in \mathbb{G}$ and $\mathbf{y}_i = C(\mathbf{u}_i, r_i) \in \mathbb{G}^d$ to the server.
3. The server randomly samples $\mathbf{a}_0 \in \mathbb{Z}_p^d$ from the uniform distribution, $\mathbf{a}_1, \ldots, \mathbf{a}_k \in \mathbb{Z}_p^d$ from the discrete normal distribution and sends $\mathbf{A}$ to the client, where $\mathbf{A}$ is the $(k+1) \times d$ matrix whose rows are $\mathbf{a}_0, \ldots, \mathbf{a}_k$.
4. The server computes $h_t = \prod_l w_l^{a_{tl}}$ for $t \in [0, k]$ and sends $\mathbf{h} = (h_0, \ldots, h_k)$ to the client.
5. The client generates a proof $\pi \leftarrow \mathsf{GenPrf}(g, q, \mathbf{w}, \mathbf{f}, b_{\text{ip}}, b_{\max}, B_0, \mathbf{A}, \mathbf{h}, z_i, r_i, \mathbf{u}_i)$ by Algorithm 3 and sends π to the server.
6. The server checks π using $\mathsf{VerPrf}(g, q, \mathbf{w}, \mathbf{f}, b_{\text{ip}}, b_{\max}, B_0, \mathbf{A}, \mathbf{h}, z_i, \mathbf{y}_i, \pi)$ in Algorithm 4.

Figure 3: Probabilistic input integrity check between the server and one client.

4 RiseFL Design

In this section, we describe our system design. We first present the rationale of the protocol in Section 4.1. Then, we introduce the hybrid commitment scheme and probabilistic integrity check method in Sections 4.2 and 4.3, respectively. Finally, we detail the protocol in Section 4.4.

4.1 Rationale

The most relevant work to our problem is EIFFeL [45], which also ensures input privacy and integrity in FL training. However, its efficiency is extremely low and, thus, is impractical to be deployed in real-world systems. For example, under the experiment settings in Section 6, given 100 clients and 1K model parameters, EIFFeL takes around 15.3 seconds for proof generation and verification on each client. More severely, the cost is increased to 152 seconds when the number of model parameters d is 10K. The underlying reason lies in that the complexity of its proof generation and verification is linearly dependent on d, making EIFFeL inefficient and less scalable. We shall detail the cost analysis of EIFFeL in Section 5.2.

The rationale behind our idea is to reduce the complexity of expensive group-exponential computations in ZKP generation and verification. To do so, we design a probabilistic L_2-norm integrity check method, as shown in Algorithm 1. The intuition is that, instead of generating and verifying proofs for the L_2-norm of an update $||\mathbf{u}||_2$, where $\mathbf{u} \in \mathbb{R}^d$, we randomly sample k points $\mathbf{a}_1, \ldots, \mathbf{a}_k$ from the normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I}_d)$. Then, the random variable:

$$\frac{1}{||\mathbf{u}||_2^2} \sum_{i=1}^{k} \langle \mathbf{a}_i, \mathbf{u} \rangle^2 \tag{1}$$

follows a chi-square distribution χ_k^2 with k degrees of freedom (see Lemma 1). In Algorithm 1, if $||\mathbf{u}||_2 \leq B$, then the probability that $\mathbf{u}$ passes the check is at least $1 - \varepsilon$ (Lemma 4), where ε is chosen to be cryptographically small, e.g. 2^{-128}. In this way, the probability that the client fails the check is of the same order as the probability that the client's encryption is broken. We shall present the detailed constructions of proof generation and verification based on this method in Section 4.3. Moreover, we propose a hybrid commitment scheme to efficiently support this probabilistic check method, as will be presented in Section 4.2. As a consequence, we can significantly reduce the cryptographic operation costs from $O(d)$ to $O(d/\log d)$.

4.2 Hybrid Commitment Scheme

We first introduce our hybrid commitment scheme that will be used in the probabilistic check method. After each client $\mathcal{C}_i (i \in [n])$ computes the local update $\mathbf{u}_i$, it first needs to commit $\mathbf{u}_i$ to the server before generating the proofs. Assume that the clients and server agree on independent group elements $g, w_1, \ldots, w_d \in \mathbb{G}$, where $w_j (j \in [d])$ is used for committing the j-th coordinate in $\mathbf{u}_i$. Then, $\mathcal{C}_i$ generates a random secret $r_i \in \mathbb{Z}_p$ and encrypts $\mathbf{u}_i$ with Pedersen commitment as follows:

$$\begin{aligned} C(\mathbf{u}_i, r_i) &= (C(u_{i1}, r_i), \ldots, C(u_{id}, r_i)) \\ &= (g^{u_{i1}} w_1^{r_i}, \ldots, g^{u_{id}} w_d^{r_i}), \end{aligned} \tag{2}$$

where u_{ij} is the j-th coordinate in $\mathbf{u}_i$. Each client $\mathcal{C}_i$ sends $\mathbf{y}_i = C(\mathbf{u}_i, r_i)$ and $z_i = g^{r_i}$ to the server as commitments. Given that r_i is held by each client $\mathcal{C}_i$, the server knows nothing regarding each update $\mathbf{u}_i$.

To facilitate the server to aggregate well-formed updates, we also require each client $\mathcal{C}_i$ to share its secret r_i with other clients using VSSS. Specifically, $\mathcal{C}_i$ computes $((1, r_{i1}), \ldots, (n, r_{in}), \Psi_{r_i}) \leftarrow \mathsf{SS.Share}(r_i, n, m+1, g)$ and sends $((j, r_{ij}), \Psi_{r_i})$ to $\mathcal{C}_j$. Note that $g^{r_i} = \Psi_{r_i}(0)$. The secret r_i allows the server to correctly aggregate the updates from honest clients. Let $\mathcal{C}_H^*$ be the set of honest clients identified by the server and clients (we will discuss how the server and clients collaboratively identify this set in Section 4.4). The server can compute $\mathcal{U} = \sum_{\mathcal{C}_i \in \mathcal{C}_H^*} \mathbf{u}_i$ as follows. First, the

Algorithm 2 $\mathsf{VerCrt}(\mathbf{w},\mathbf{h},\mathbf{A})$

Input: $\mathbf{w}=(w_1,\dots,w_d)\in\mathbb{G}^d$, $\mathbf{h}=(h_0,\dots,h_k)\in\mathbb{Z}_p^{(k+1)}$, $\mathbf{A}\in\mathbb{M}_{(k+1)\times d}(\mathbb{Z}_p)$.
Randomly Sample $\mathbf{b}=(b_0,\dots,b_k)\in\mathbb{Z}_p^{k+1}$.
Compute $\mathbf{c}=(c_1,\dots,c_d)=\mathbf{b}\cdot\mathbf{A}\in\mathbb{Z}_p^d$.
return $h_0^{b_0}\dots h_k^{b_k}==w_1^{c_1}\dots w_d^{c_d}$.

server aggregates the commitments from $\mathcal{C}_H^*$ by:

$$\begin{aligned} C(\mathcal{U},r) &= \left(\prod_{\mathcal{C}_i\in\mathcal{C}_H^*}C(u_{i1},r_i),\dots,\prod_{\mathcal{C}_i\in\mathcal{C}_H^*}C(u_{id},r_i)\right)\\ &= \left(g^{\sum_{\mathcal{C}_i\in\mathcal{C}_H^*}u_{i1}}w_1^{\sum_{\mathcal{C}_i\in\mathcal{C}_H^*}r_i},\dots,g^{\sum_{\mathcal{C}_i\in\mathcal{C}_H^*}u_{id}}w_d^{\sum_{\mathcal{C}_i\in\mathcal{C}_H^*}r_i}\right)\\ &= (g^{u_1}w_1^r,\dots,g^{u_d}w_d^r), \end{aligned} \tag{3}$$

where $r=\sum_{\mathcal{C}_i\in\mathcal{C}_H^*}r_i$. Note that r can be computed by the clients in $\mathcal{C}_H^*$ using secure aggregation. That is, for each client $\mathcal{C}_i\in\mathcal{C}_H^*$, it sums the secret shares $r_i'=\sum_{\mathcal{C}_j\in\mathcal{C}_H^*}r_{ji}$ and sends it to the server. The server checks the integrity of each r_i' against $\prod_{\mathcal{C}_j\in\mathcal{C}_H^*}\Psi_{r_j}$, and uses the ones that pass the check to recover r'. According to the homomorphic property of VSSS, $r'=r$. Consequently, with the knowledge of r, the server computes g^{u_l} for $l\in[d]$ and solves u_l according to Eqn 3, which is the aggregation of the l-coordinate in honest clients' updates.

4.3 Probabilistic Input Integrity Check

Next, we present how the server and clients execute the probabilistic integrity check. Suppose the server and clients agree on some necessary parameters, including the number of samples k in Eqn 1 for the probabilistic L_2-norm check. We will discuss the effect of the choice of k in Section 5.1.

Without loss of generality, we describe the check for one client $\mathcal{C}_i$, as summarized in Figure 3. The client first sends the commitments $z_i=g^{r_i}$ and $\mathbf{y}_i=C(\mathbf{u}_i,r_i)$ to the server using the hybrid commitment scheme. Then, the server randomly generates $k+1$ random samples, say $\mathbf{a}_0,\dots,\mathbf{a}_k\in\mathbb{Z}_p^d$. $\mathbf{a}_0$ is sampled from the uniform distribution on $\mathbb{Z}_p$ with cryptographically secure pseudo-random number generator (PRNG) for checking the integrity of Pedersen commitments $\mathbf{y}_i$. $\mathbf{a}_1,\dots,\mathbf{a}_k$ are sampled from the discrete normal distribution with insecure PRNG for fast execution of probabilistic check in Algorithm 1. After that, the server computes $h_t=\prod_l w_l^{a_{tl}}$ for $t\in[0,k]$. Let $\mathbf{A}=(\mathbf{a}_0,\mathbf{a}_1,\cdots,\mathbf{a}_k)$ and $\mathbf{h}=(h_0,h_1,\cdots,h_k)$. The server sends $\{\mathbf{A},\mathbf{h}\}$ to the client.

Upon receiving the information, the client generates the proof π using Algorithm 3. Specifically, the client first verifies the correctness of $\mathbf{h}$ using $\mathsf{VerCrt}(\mathbf{w},\mathbf{h},\mathbf{A})$ from Algorithm 2. This is to ensure that the server does not steal information from the client by sending incorrect $\mathbf{h}$. Note that Algorithm 2 uses batch verification to accelerate the verification of $h_t=\prod_l w_l^{A_{tl}}$ for $t\in[0,k]$. If it is correct, the client computes the following items for generating the proof that Eqn 1 is less than a bound, based on Algorithm 1.

Algorithm 3 $\mathsf{GenPrf}(g,q,\mathbf{w},\mathbf{f},b_{\mathsf{ip}},b_{\max},B_0,\mathbf{A},\mathbf{h},z,r,\mathbf{u})$

Input: $g,q,z\in\mathbb{G}$, $\mathbf{w}\in\mathbb{G}^d$, $\mathbf{f}\in\mathbb{G}^{2kb_{\max}}$, $b_{\mathsf{ip}}<b_{\max}$, $B_0\in(0,2^{b_{\max}}]$, $A\in\mathbb{M}_{(k+1)\times d}(\mathbb{Z}_p)$, $\mathbf{h}\in\mathbb{G}^{(k+1)}$, $r\in\mathbb{Z}_p$, $\mathbf{u}\in\mathbb{Z}_p^d$.
if not $\mathsf{VerCrt}(\mathbf{w},\mathbf{h},\mathbf{A})$ **then**
  Abort
end if
Compute $\mathbf{v}^*=\mathbf{u}\cdot\mathbf{A}^T\in\mathbb{Z}_p^{k+1}$. Denote $\mathbf{v}^*=(v_0,\dots,v_k)$, and let $\mathbf{v}=(v_1,\dots,v_k)$.
Compute $e_t=g^{v_t}h_t^r$ for $t\in[0,k]$, and let $\mathbf{e}=(e_0,e_1,\dots,e_k)$.
Randomly sample $\mathbf{s},\mathbf{s}'\in\mathbb{G}^k$.
Compute $o_t=g^{v_t}q^{s_t}$, $o_t'=g^{v_t^2}q^{s_t'}$ for $t\in[1,k]$.
Compute $\rho\leftarrow\mathsf{GenPrfWf}(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},r,\mathbf{v}^*,\mathbf{s})$.
Compute $\tau\leftarrow\mathsf{GenPrfSq}(g,q,\mathbf{o},\mathbf{o}',\mathbf{v},\mathbf{s},\mathbf{s}')$.
Compute $\sigma\leftarrow\mathsf{GenPrfBd}(g,q,\mathbf{f},b_{\mathsf{ip}},g^{2^{b_{\mathsf{ip}}-1}}\cdot\mathbf{o},\mathbf{v}+2^{b_{\mathsf{ip}}-1}\cdot\mathbf{1},\mathbf{s})$.
Compute $\mu\leftarrow\mathsf{GenPrfBd}(g,q,\mathbf{f},b_{\max},g^B(\prod_{t=1}^k o_t')^{-1},B-\sum_{t=1}^k v_t^2,-\sum_{t=1}^k s_t')$.
return $\pi=(\mathbf{e},\mathbf{o},\mathbf{o}',\rho,\tau,\sigma,\mu)$.

Algorithm 4 $\mathsf{VerPrf}(g,q,\mathbf{w},\mathbf{f},b_{\mathsf{ip}},b_{\max},B_0,\mathbf{A},\mathbf{h},z,\mathbf{y},\pi)$

Input: $g\in\mathbb{G}$, $q\in\mathbb{G}$, $\mathbf{w}\in\mathbb{G}^d$, $\mathbf{f}\in\mathbb{G}^{2kb_{\max}}$, $b_{\mathsf{ip}}<b_{\max}$, $B_0\in(0,2^{b_{\max}}]$, $A\in\mathbb{M}_{(k+1)\times d}(\mathbb{Z}_p)$, $\mathbf{h}\in\mathbb{G}^k$, $\mathbf{y}\in\mathbb{G}^d$, π.
Unravel $\pi=(\mathbf{e},\mathbf{o},\mathbf{o}',\rho,\tau,\sigma,\mu)$.
return $\mathsf{VerCrt}(\mathbf{y},\mathbf{e},\mathbf{A})$ **and**
  $\mathsf{VerPrfWf}(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},\rho)$ **and**
  $\mathsf{VerPrfSq}(g,q,\mathbf{o},\mathbf{o}',\tau)$ **and**
  $\mathsf{VerPrfBd}(g,q,\mathbf{f},b_{\mathsf{ip}},g^{2^{b_{\mathsf{ip}}-1}}\cdot\mathbf{o},\sigma))$ **and**
  $\mathsf{VerPrfBd}(g,q,\mathbf{f},b_{\max},g^B(\prod_{t=1}^k o_t')^{-1},\mu)$.

- The client computes the inner products between $\mathbf{u}_i$ and each row of $\mathbf{A}$, obtaining $\mathbf{v}^*=(v_0,v_1,\cdots,v_k)$, where $v_t=\langle\mathbf{a}_t,\mathbf{u}_i\rangle$ for $t\in[0,k]$. The client commits $e_t=g^{v_t}h_t^{r_i}$ using its secret r_i for $t\in[0,k]$. Let $\mathbf{e}^*=(e_0,e_1,\cdots,e_k)$ and $\mathbf{e}=(e_1,\cdots,e_k)$. The commitment e_0 is used for integrity check of $\mathbf{y}_i$. The commitments $\mathbf{e}$ are used for bound check of $\mathbf{v}$.
- Let $\mathbf{v}=(v_1,\cdots,v_k)$. The client commits v_t using $o_t=g^{v_t}q^{s_t}$ for $t\in[1,k]$, where s_t is a random number. Let $\mathbf{o}=(o_1,\cdots,o_k)$ be the resulted commitment. e_t and o_t commit to the same secret v_t using different group elements h_t and q. As a result, $o_1,\dots,o_k$ use the same group element q, ready for batch square checking and batch bound checking.
- The client further commits $o_t'=g^{v_t^2}q^{s_t'}$ for $t\in[1,k]$, where s_t' is a random number. Let $\mathbf{o}'=(o_1',\cdots,o_k')$ be the resulted commitment. This commitment will be used in the proof generation and verification for proof of square.
- The client generates a proof ρ to prove that $(z,\mathbf{e}^*,\mathbf{o})$ is well-

formed, which means that the secret in z is used as the blind in e_t, $t \in [0,k]$, and that the secrets in e_t and o_t are equal, $t \in [1,k]$. Note that $z = g^{r_i} = \Psi_{r_i}(0)$ is the 0-th coordinate of the check string of Shamir's share of r_i.

- The client generates a proof τ to prove that the secret in o'_t is the square of the secret in o_t for $t \in [1,k]$, using the building block described in Section 2.1.
- The client generates a proof σ that the secret in o_t is in the interval $[-2^{b_{\mathsf{ip}}}, 2^{b_{\mathsf{ip}}})$ for $t \in [1,k]$. This ensures the inner product of $\mathbf{a}_t$ and $\mathbf{u}_i$ does not cause overflow when squared.
- The client generates a proof μ that $B - \sum_t v_t^2$ is in the interval $[0, 2^{b_{\mathsf{max}}})$ using the commitment $g^B(\prod_t o'_t)^{-1}$. This proof is to guarantee that Eqn 1 is less than the bound of the probabilistic check.

As a result, the client sends the proof $\pi = (\mathbf{e}, \mathbf{o}, \mathbf{o}', \rho, \tau, \sigma, \mu)$ to the server. After receiving the proof, the server can verify it accordingly, including: checking the correctness of $e_t = \prod_l y_{il}^{a_{tl}}$, $t \in [0,k]$ using Algorithm 2, checking the well-formedness proof ρ using Algorithm 8 in Appendix A.1, checking the square proofs of $(\mathbf{o}', \mathbf{o})$, and checking the two bound proofs. If all the checks are passed, the server guarantees that the client's update passes the check in Algorithm 1.

4.4 Protocol Description

We present the full protocol of RiseFL in Figure 4, including a system initialization stage, followed by three iterative rounds.

System Initialization. All parties are given the system parameters, including the number of clients n, the maximum number of malicious clients m, the bound on the number of bits b_{ip} of each inner product, the maximum number of bits $b_{\max} > b_{\text{ip}}$ of the sum of squares of inner product, the bound of the sum of inner products $B_0 < 2^{b_{\max}}$, the number of samples k for the probabilistic check, a set of independent group elements $g \in \mathbb{G}, q \in \mathbb{G}, \mathbf{w} \in \mathbb{G}^d, \mathbf{f} \in \mathbb{G}^{2kb_{\max}}$, the factor $M > 0$ used in discretizing the normal distribution samples, and a cryptographic hash function $H(\cdot)$. Since there is no direct channel between any two clients, we let the server forward some of the messages. To prevent the server from accessing the secret information, each client $\mathcal{C}_i (i \in [n])$ generates a public/private key pair (pk_i, sk_i) and sends the public key pk_i to a public bulletin. Then, each client fetches the other clients' public keys such that each pair of clients can establish a secure channel via the Diffie-Hellman protocol [37] for exchanging messages.

Round 1: Commitment Generation. In every FL training iteration, each client $\mathcal{C}_i$ generates a random number r_i as the secret. Then, $\mathcal{C}_i$ adopts the hybrid commitment scheme in Section 4.2 to commit its update $\mathbf{u}_i$ using Eqn 2, obtaining $\mathbf{y}_i = C(\mathbf{u}_i, r_i)$. Also, it generates the secret shares of r_i using VSSS, obtaining $((1, r_{i1}), \ldots, (n, r_{in}), \Psi_{r_i})$, and encrypts each share r_{ij} ($\forall j \in [1,n] \wedge j \neq i$) using the encryption key based on (pk_j, sk_i). Next, $\mathcal{C}_i$ sends the encrypted shares $\mathsf{Enc}(r_{ij})$ and the check string Ψ_{r_i} to the server. Afterward, the server forwards the encrypted shares to respective clients and broadcasts the check strings to all the clients. In addition, the server initializes a list $\mathcal{C}^* = \varnothing$ for the current iteration to record the malicious clients that will be identified in the following round.

Round 2: Proof Generation and Verification. In this round, the clients and the server jointly flag the malicious clients in two steps. The first step is to verify the authenticity of secret shares. After receiving the encrypted shares $\mathsf{Enc}(r_{ji})$ and check strings Ψ_{r_j} for $j \in [1,n]$, each client $\mathcal{C}_i$ decrypts r_{ji} and checks against Ψ_{r_j}. Then, $\mathcal{C}_i$ sends a list of candidate malicious clients that do not pass the check to the server. The server follows two rules to flag the malicious clients [45]: (1) if a client $\mathcal{C}_i$ flags more than m clients as malicious or is flagged as malicious by more than m clients, the server puts $\mathcal{C}_i$ to $\mathcal{C}^*$; (2) if a client $\mathcal{C}_i$ is flagged as malicious by $[1,m]$ clients, the server requests the shares r_{ij} in the clear for all $\mathcal{C}_j$ that flags $\mathcal{C}_i$ and checks against Ψ_{r_i}, if the clear r_{ij} fails the check, the server puts $\mathcal{C}_i$ to $\mathcal{C}^*$. This ensures that if $\mathcal{C}_i$ is honest, then $\mathcal{C}_i$ passes the check and its clear r_{ij} is sent only for malicious $\mathcal{C}_j$ and at most m clear text shares are sent.

The second step is to verify the integrity of each client's update using the probabilistic method. Note that in Section 4.3, we require the server to send $k+1$ random samples with dimensionality d to the client. To reduce the communication cost, we let the server select a random value s and broadcast it to all clients. Based on s, the server and each client $\mathcal{C}_i (i \in [n])$ first compute a seed $H(s, \{pk_i\}_{1 \le i \le n})$ using s and all clients' public keys. Hence, the clients and the server can generate the same set of random samples $\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_k)$ because the seed is the same. Then, the server computes $\mathbf{h}$ (see Section 4.3) and broadcasts it to all clients. Next, each client $\mathcal{C}_i (i \in [1,n])$ generates $\mathbf{A}$, computes the proof π_i according to Algorithm 3, and sends π_i to the server. Consequently, the server verifies the proof of each client $\mathcal{C}_i$ and puts it to $\mathcal{C}^*$ if the verification fails. The list $\mathcal{C}^*$ is broadcast to all the clients.

Round 3: Aggregation. In this round, each client $\mathcal{C}_i$ selects the corresponding shares from $\mathcal{C}_j \notin \mathcal{C}^*$, aggregates them to get r'_i, and sends r'_i to the server. The server uses $\mathsf{SS.Verify}$ to verify the integrity of each r'_i and uses $\mathsf{SS.Recover}$ with the ones that pass the integrity check to recover $\sum_{\mathcal{C}_j \notin \mathcal{C}^*} r_j$. Therefore, the server can solve the equation in Eqn 3 to compute the aggregation of honest clients' updates.

5 Analysis

5.1 Security Analysis

Theorem 1. *By choosing $\varepsilon = \text{negl}(\kappa)$ and $B_0 = B^2M^2(\sqrt{\gamma_{k,\varepsilon}} + \frac{\sqrt{kd}}{2M})^2$, for any list of honest clients $\mathcal{C}_H$ of size at least $n-m$,*

- **System Initialization.**
  - All parties are given the client number n, the maximum malicious client number m, the bit number b_{ip} for inner products, the maximum bit number $b_{\mathsf{max}} > b_{\mathsf{ip}}$, the sum bound of the inner products $B_0 < 2^{b_{\mathsf{max}}}$, the sample number k, group elements $g, q \in \mathbb{G}$, $\mathbf{w} \in \mathbb{G}^d$ and $\mathbf{f} \in \mathbb{G}^{2kb_{\mathsf{max}}}$, the factor M used in discretizing the normal distribution samples.

    Each client $\mathcal{C}_i (i \in [1, n])$:
  - Generates a key pair (pk_i, sk_i) and sends pk_i to the public Bulletin and fetches other clients' public keys.
- **Round 1 (Commitment Generation).**

    Each client $\mathcal{C}_i (i \in [1, n])$:
  - Generates a random secret r_i and commits its update $\mathbf{u}_i$ with $\mathbf{y}_i = C(\mathbf{u}_i, r_i) = (g^{u_{i1}} w_1^{r_i}, \ldots, g^{u_{id}} w_d^{r_i})$.
  - Computes the verifiable secret shares of r_i: $\{(1, r_{i1}), \ldots, (n, r_{in}), \Psi_{r_i}\} \leftarrow \mathsf{SS.Share}(r_i, n, m+1, g)$.
  - Encrypts each share r_{ij} with the symmetric key (pk_j, sk_i) for client $\mathcal{C}_j (j \in [1, n])$.
  - Sends the commitment $\mathbf{y}_i$, the encrypted share $\mathsf{Enc}(r_{ij}) (j \in [1, n])$, and the check string Ψ_{r_i} to the server.

    Server:
  - Initializes the malicious client list $\mathcal{C}^* = \varnothing$, sends the encrypted shares $\mathsf{Enc}(r_{ij}) (i \in [1, n])$ to client $\mathcal{C}_j$, and broadcasts the check strings $\Psi_{r_i} (i \in [1, n])$ to all clients.
- **Round 2 (Proof Generation and Verification).**

  (i) Verify the authenticity of secret shares:

    Each client $\mathcal{C}_i (i \in [1, n])$:
  - Downloads the check strings Ψ_{r_j} and the encrypted $\mathsf{Enc}(r_{ij})$ for $j \in [1, n]$ from the server.
  - Decrypts r_{ji} using (pk_j, sk_i), checks r_{ji} against Ψ_{r_j}, and sends the list of clients that fail the check to the server.

    Server:
  - If client $\mathcal{C}_i$ flags more than m clients or more than m clients flag client $\mathcal{C}_i$, puts $\mathcal{C}_i$ into $\mathcal{C}^*$.
  - For a client $\mathcal{C}_i$ that receives $[1, m]$ flags, the server requests r_{ij} (in clear) for all $\mathcal{C}_j$ that flags $\mathcal{C}_i$ and checks them against Ψ_{r_i}. If any of the clear r_{ij} does not pass the check, puts $\mathcal{C}_i$ into $\mathcal{C}^*$; else, sends r_{ij} (in clear) to $\mathcal{C}_j$.

  (ii) Verify the L_2-norm of each client's update:

    Server:
  - Generates a random number s and sends s to all the clients.
  - Randomly samples $\mathbf{a}_0 \in \mathbb{Z}_p^d$ that follows the uniform distribution using $H(s, (pk_i)_{1 \le i \le n})$ as the seed.
  - Randomly samples $\mathbf{a}_t \in \mathbb{Z}_p^d$ for $t \in [1, k]$ that follows $\mathcal{N}(0, M^2)$ and rounded to the nearest integer using $H(s, (pk_i)_{1 \le i \le n})$ as the seed. Let $\mathbf{A}$ be the $(k+1) \times d$ matrix whose rows are $\mathbf{a}_0, \ldots, \mathbf{a}_k$.
  - Computes $h_t = \prod_l w_l^{a_{tl}}$ for $t = [0, k]$ and sends $\mathbf{h} = (h_0, h_1, \ldots, h_k)$ to all the clients.

    Each client $\mathcal{C}_i$:
  - Computes $\mathbf{A}$ using $H(s, (pk_i)_{1 \le i \le n})$ as the seed.
  - Computes $\pi_i \leftarrow \mathsf{GenPrf}(g, q, \mathbf{w}, \mathbf{f}, b_{\mathsf{ip}}, b_{\mathsf{max}}, B_0, \mathbf{A}, \mathbf{h}, \Psi_{r_i}(0), r_i, \mathbf{u}_i)$, and sends π_i to the server.

    Server:
  - Uses $\mathsf{VerPrf}(g, q, \mathbf{w}, \mathbf{f}, b_{\mathsf{ip}}, b_{\mathsf{max}}, B_0, \mathbf{A}, \mathbf{h}, \Psi_{r_i}(0), \mathbf{y}_i, \pi_i)$ to verify Client i's proof. Puts i into $\mathcal{C}^*$ if check fails.
- **Round 3 (Aggregation of Clients' Updates).**

    Each client $\mathcal{C}_i (i \in [1, n])$:
  - Receives $\mathcal{C}^*$ from the server and sends the aggregated share $r'_i = \sum_{j \notin \mathcal{C}^*} r_{ji}$ to the server.

    Server:
  - Sets $\mathcal{R} = \varnothing$. Until $\mathcal{R}$ has size $m+1$: for each $\mathcal{C}_i \notin \mathcal{C}^*$, checks r'_i using $\mathsf{SS.Verify}(\prod_{\mathcal{C}_j \notin \mathcal{C}^*} \Psi_{r_j}, i, r'_i, n, m+1, g)$, puts i into $\mathcal{R}$ if check passes. Reconstructs $r' \leftarrow \mathsf{SS.Recover}(\{(i, r'_i) : i \in \mathcal{R}\})$.
  - For each $l \in [1, d]$, computes $g^{\sum_{\mathcal{C}_i \notin \mathcal{C}^*} u_{il}} = w_l^{-r'} \prod_{\mathcal{C}_i \notin \mathcal{C}^*} y_{il}$ and solves $\sum_{\mathcal{C}_i \notin \mathcal{C}^*} u_{il}$.

Figure 4: The overall description of the proposed RiseFL protocol.

RiseFL satisfies $(D, F_{k,\varepsilon,d,M})$*-SAVI, where* $D(\boldsymbol{u}) = \|\boldsymbol{u}\|_2/B$ *and*

$$F_{k,\varepsilon,d,M}(c) = \Pr_{x\sim\chi_k^2}\left[x < \frac{1}{c^2}\left(\sqrt{\gamma_{k,\varepsilon}} + \frac{3\sqrt{kd}}{2M}\right)^2\right] + \mathrm{negl}(\kappa). \quad (4)$$

The security proof is composed of several parts. In the following, we outline the proof structure and defer the complete proof to Appendix A.3.4.

Lemma 1 states that the sum of inner products of normal distribution samples follows the chi-square distribution. Lemmas 2-3 bound the rounding errors occurring in discretizing the normal distribution samples.

Lemma 1. *Suppose that* $\boldsymbol{u} \in \mathbb{R}^d \setminus \{\boldsymbol{0}\}$ *and* $\boldsymbol{a}_1, \dots, \boldsymbol{a}_k$ *are sampled i.i.d. from the normal distribution* $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_d)$*. Then,* $\frac{1}{\|\boldsymbol{u}\|_2^2}\sum_{t=1}^k \langle \boldsymbol{a}_t, \boldsymbol{u}\rangle^2$ *follows the chi-square distribution* χ_k^2*.*

Proof. In Appendix A.2.1. □

Lemma 2. *Suppose that* $\boldsymbol{u} \in \mathbb{R}^d$ *and* $\|\boldsymbol{u}\|_2 \le B$*. Define* $\mathsf{round}: \mathbb{R} \to \mathbb{Z}$ *by* $\mathsf{round}(n+\alpha) = n$ *for* $n \in \mathbb{Z}$, $-1/2 \le \alpha < 1/2$*. Suppose that* $\boldsymbol{b}_1, \dots, \boldsymbol{b}_k \in \mathbb{R}^d$ *satisfy that*

$$\sum_{t=1}^k \langle \boldsymbol{b}_t, \boldsymbol{u}\rangle^2 \le B^2M^2r. \quad (5)$$

Define $\boldsymbol{a}_t$ *by* $a_{tj} = \mathsf{round}(b_{tj})$ *for* $t \in [1,k]$*. Then*

$$\sum_{t=1}^k \langle \boldsymbol{a}_t, \boldsymbol{u}\rangle^2 \le B^2M^2\left(\sqrt{r} + \frac{\sqrt{kd}}{2M}\right)^2. \quad (6)$$

Proof. In Appendix A.2.2. □

Lemma 3. *Suppose that* $\boldsymbol{u} \in \mathbb{R}^d$ *and* $\|\boldsymbol{u}\|_2 > B$*. Define* round *as in Lemma 2. Suppose that* $\boldsymbol{b}_1, \dots, \boldsymbol{b}_k \in \mathbb{R}^d$ *and* $\boldsymbol{a}_t$ *is defined by* $a_{tj} = \mathsf{round}(b_{tj})$ *for* $t \in [1,k]$*. Suppose that* $\sum_{t=1}^k \langle \boldsymbol{a}_t, \boldsymbol{u}\rangle^2 \le B_0$*. Then,*

$$\frac{1}{\|\boldsymbol{u}\|_2^2}\sum_{t=1}^k \langle \boldsymbol{b}_t, \boldsymbol{u}\rangle^2 \le \frac{M^2}{D(\boldsymbol{u}_i)^2}\left(\sqrt{\gamma_{k,\varepsilon}} + \frac{3\sqrt{kd}}{2M}\right)^2. \quad (7)$$

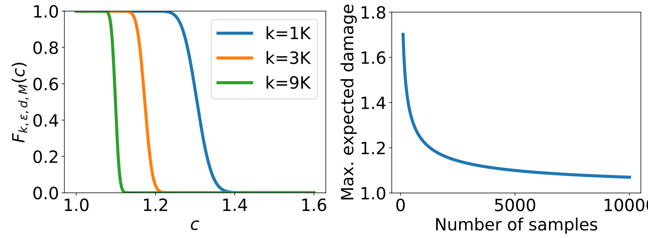
Proof. In Appendix A.2.3. □

We prove input integrity and input privacy separately. For integrity, we need to prove that: (1) an honest client has a negligible failure rate (Lemma 4); (2) a malicious client's pass rate is bounded by $F_{k,\varepsilon,d,M}$ (Lemmas 5-6); and (3) the output is the correct aggregation (Lemma 7).

Lemma 4. *If* $\mathcal{C}_i$ *is honest, then the probability that* $\boldsymbol{u}_i$ *passes the integrity check is at least* $1-\varepsilon$*.*

Proof. In Appendix A.3.1. □

Lemma 5. *The probability of* $\mathcal{C}_i$ *passing the integrity check without committing to* $\boldsymbol{u}_i$ *satisfying* $\sum_{t=1}^k \langle \boldsymbol{a}_t, \boldsymbol{u}_i\rangle^2 \le B_0$ *is* $\mathrm{negl}(\kappa)$.



(a) k vs. $F_{k,\varepsilon,d,M}$ (b) k vs. maximum damage

Figure 5: The effect of k when $\varepsilon = 2^{-128}$, $d = 10^6$, $M = 2^{24}$: (a) the trend of $F_{k,\varepsilon,d,M}$ w.r.t. k; (b) the trend of the maximum damage w.r.t. k.

Proof. In Appendix A.3.2. □

Lemma 6. *If* $\mathcal{C}_i$ *is malicious with* $\|\boldsymbol{u}_i\|_2 > B$*, then the probability that* $\boldsymbol{u}_i$ *passes the integrity check is at most* $F_{k,\varepsilon,d,M}(D(\boldsymbol{u}_i))$*.*

Proof. In Appendix A.3.3. □

Lemma 7. *With probability at most* $1-\mathrm{negl}(\kappa)$*, the protocol outputs* $\sum_{\mathcal{C}_i \in \mathcal{C}_{\mathsf{Valid}}} \boldsymbol{u}_i$ *where* $\mathcal{C}_H \subseteq \mathcal{C}_{\mathsf{Valid}}$*.*

Proof. By Lemma 4, the probability that every honest client passes the integrity check is at least $1-\mathrm{negl}(\kappa)$. The probability that at least one malicious client breaks VSSS is $\mathrm{negl}(\kappa)$. If VSSS is not broken and every honest client passes the integrity check, the protocol outputs $\sum_{\mathcal{C}_i \in \mathcal{C}_{\mathsf{Valid}}} \mathbf{u}_i$ and $\mathcal{C}_H \subseteq \mathcal{C}_{\mathsf{Valid}}$ as Shamir's secret sharing is additively homomorphic. □

In terms of input privacy, we prove in Lemma 8 that nothing except the aggregation of honest updates can be learned.

Lemma 8. *If* $\mathcal{C}_H$ *is a list of honest clients with size at least* $n-m$*, then with probability at least* $1-\mathrm{negl}(\kappa)$*, nothing except* $\sum_{\mathcal{C}_i \in \mathcal{C}_H} \boldsymbol{u}_i$ *is revealed from the outputs of* $\mathcal{C}_H$*.*

Proof. Cryptographically, the values of Ψ_{r_i}, $C(u_i, r_i)$, and π_i do not reveal any information about x_i or r_i. VSSS ensures that nothing is revealed from the $\le m$ shares $\{r_{ij}\}_{j \notin \mathcal{C}_H}$ of the secret r_i. At Round 3, VSSS ensures that only $\sum_{\mathcal{C}_i \in \mathcal{C}_H} r_i$ is revealed. From $\sum_{\mathcal{C}_i \in \mathcal{C}_H} r_i$, the only value that can be computed is $\sum_{\mathcal{C}_i \in \mathcal{C}_H} \mathbf{u}_i$. □

Discussion. Now we discuss the effect of k on $F_{k,\varepsilon,d,M}$ and the maximum damage, as illustrated in Figure 5. We set $\varepsilon = 2^{-128}$ by default in this paper. We set $M = 2^{24}$ so that when $k \le 10^4$ and $d \le 10^6$, the term $\frac{3\sqrt{kd}}{2M}$ is insignificant in Eqn 4. Figure 5a shows the trend of $F_{k,\varepsilon,d,M}$ with $d = 10^6$ and different choices of k. We can see that $F_{k,\varepsilon,d,M}(c)$ is very close to 1 when c is slightly bigger than 1, and drops rapidly to negligible as c continues to increase. That means for instance, when $k = 1000$, a malicious $\mathbf{u}_i$ with $\|\mathbf{u}_i\|_2 \le 1.2B$ will very likely

Table 1: Cost comparison (g.e. = group exponentiation, f.a. = field arithmetic)

		EIFFeL		RoFL		RiseFL	
		g.e.	f.a.	g.e.	f.a.	g.e.	f.a.
Client comp.	commit.	$O(md)$	$O(nmd)$	$O(d)$	small	$O(d)$	small
	proof gen.	0	$O(bnmd)$	$O(db)$	small	$O(d/\log d)$	$O(kd)$
	proof ver.	$O(nmd/\log(md))$	$O(bnmd)$	0	0	0	0
	total	$O(md(1+n/\log(md)))$	$O(bnmd)$	$O(db)$	small	$O(d)$	$O(kd)$
Server comp.	prep.	0	0	0	0	$O(kd\log M/\log d\log p)$	small
	proof ver.	0	small	$O(ndb/\log(db))$	small	$O(nd/\log d)$	small
	agg.	0	$O(nmd)$	$O(nd/\log p)$	small	$O(nd/\log p)$	small
	total	0	$O(nmd)$	$O(ndb/\log(db))$	small	$O(d(n+k\log M/\log p)/\log d)$	small
Comm. per client		$\approx 2dnb$ elements		$\approx 12d$ elements		$\approx d$ elements	

pass the integrity check, but when $||\mathbf{u}_i||_2 \geq 1.4B$, the check will fail with close-to-1 probability.

This is the downside of the probabilistic L_2-norm checking: slightly out-of-bound vectors may pass the check. We can quantify the size of damage caused by such malicious vectors. For example, with strict checking, a malicious client can use a malicious $\mathbf{u}$ with $||\mathbf{u}||_2 = B$, which passes the check, so it can do damage of magnitude B to the aggregate. With the probabilistic check, the malicious client can use slightly larger $||\mathbf{u}||_2$ at a low failure rate. The expected damage it can do to the aggregate is $||\mathbf{u}||_2 \cdot F_{k,\varepsilon,d,M}(||\mathbf{u}||_2/B)$, and by choosing a suitable $||\mathbf{u}||_2$, the maximum expected damage is

$$B \cdot \max\{c \cdot F_{k,\varepsilon,d,M}(c) : c \in (1, +\infty)\}. \tag{8}$$

Figure 5b shows the maximum expected damage with respect to k when $B = 1$. It turns out that with $\varepsilon = 2^{-128}$ and $k \geq 10^3$, the maximum expected damage is close to 1. In other words, the magnitude of damage that a malicious client can do to the aggregate is only slightly more than the one under the strict L_2-norm check protocol. We will experiment with $k = 1\text{K}, 3\text{K}, 9\text{K}$ in Section 6, corresponding to the ratio of the magnitudes of damages 1.24, 1.13, 1.08, respectively.

5.2 Cost Analysis

We theoretically analyze the cost of RiseFL under the assumption that $d >> k$ by comparing it to EIFFeL and RoFL, as summarized in Table 1, where we count the number of cryptographic group exponentiations (g.e.) and finite field arithmetic (f.a.) separately.

EIFFeL. The commitment includes the Shamir secret shares and the check string for each coordinate. The Shamir shares incur a cost of $O(nmd)$ f.a. operations. The check strings use information-theoretic secure VSSS[2] [42], which involves $O(m)$ group exponentiations per coordinate for Byzantine tolerance of m malicious clients. The proof generation[3] and verification[4] (excluding verification of check strings) costs $O(bnmd)$ f.a. and about $2dnb$ elements of bandwidth on the client side, where b is the bit length of the update. The verification of the check strings of one client takes a mult-exponentiation of length $(m+1)d$, or $O(md/\log md)$ g.e. using a Pippenger-like algorithm [43]. The server cost is small compared to client cost.

RoFL. It uses the ElGammal [22] commitment $(g^{u_{il}}h^{r_{il}}, g^{r_{il}})$ for each coordinate u_{il} with a separate blind r_{il}, which costs $O(d)$ g.e. in total. The dominating cost of proof generation includes the generation of a well-formedness proof, which involves $O(d)$ g.e., $4d$ elements, a commitment and a proof of squares ($O(d)$ g.e., $6d$ elements), and a proof of bound of each coordinate ($4\log(bd)$ multi-exponentiations of length bd, or $O(bd)$ g.e.). The proof verification is executed by the server, where the verification of the bound proof per client takes 1 multi-exponentiation of length about $2bd$. The aggregation cost is small.

RiseFL (Ours). We use Pederson commitment $g^{u_{il}}w_i^{r_i}$ for each coordinate, which costs $O(d)$ g.e. in total. The main cost of ZKP is the sub-protocol in Figure 3. On the client side, the main cost of proof generation is to verify the correctness of $\mathbf{h}$ received from the server, using VerCrt. It costs one multi-exponentiation of length d, or $O(d/\log d)$ g.e., plus $O(kd)$ f.a. to compute $\mathbf{b} \cdot \mathbf{A}$. On the server side, the main cost can be divided into two parts: (1) the computation of multi-exponentiations $\mathbf{h}$ at the preparation stage; (2) the verification of the correctness of $\mathbf{e}$ for each client using VerCrt at the proof verification stage. Each h_t, $t \in [1, k]$ is a multi-exponentiation of length d where the powers are discrete normal samples

[2] The Feldman check string [19] is not secure because weight updates are small. u_{il} can be easily computed from $g^{u_{il}}$ because u_{il} has short bit-length.

[3] In order to prevent overflow in finite field arithmetic, client $\mathcal{C}_i$ has to prove that each u_{il} is bounded, so it has to compute shares of every bit of u_{il}.

[4] We use the multiplicative homogeneity of Shamir's share to compute shares of the sum of squares at the cost of requiring that $m < (n-1)/4$. This is discussed in [45, Section 11.1]. The corresponding cost is $O(d)$ per sum of squares. In comparison, the polynomial interpolation approach in [45, Section 11.1] is actually $O(d^2)$ per sum of squares, because one needs to compute d values of polynomials of degree d, even if the Lagrange coefficients are precomputed.

Table 2: Breakdown cost comparison w.r.t. the number of model parameters d, where $k = 1000$

#Param.	Approach	Client Computation (seconds)				Server Computation (seconds)				Comm. Cost per Client (MB)
		commit.	proof gen.	proof ver.	total	prep.	proof ver.	agg.	total	
$d = 1$K	EIFFeL	0.865	3.63	11.7	16.2	-	-	0.182	**0.182**	125
	RoFL	0.051	4.43	-	4.5	-	91.2	0.040	91.3	**0.37**
	RiseFL (ours)	0.054	1.48	0.08	**1.6**	1.17	75.6	0.071	76.8	0.44
$d = 10$K	EIFFeL	8.38	36.8	115	161	-	-	1.81	**1.81**	1250
	RoFL	0.51	46.4	-	46.9	-	860	0.41	860	3.66
	RiseFL (ours)	0.49	1.8	0.08	**2.3**	8.61	82.5	0.71	91.8	**0.71**
$d = 100$K	EIFFeL	84.7	382	1070	1536	-	-	18.8	**18.8**	12500
	RoFL	5.1	496	-	502	-	8559	4.1	8563	36.6
	RiseFL (ours)	4.8	4.5	0.08	**9.3**	73.3	139	7.2	219	**3.5**
$d = 1$M	EIFFeL	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	RoFL	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	RiseFL (ours)	48.0	31.2	0.08	**79.3**	653	612	72.1	**1338**	**30.9**

from $\mathcal{N}(0, M^2)$, whose bit-lengths are $\log(M)$. The cost of computing such an h_t is $O(d \log M / \log d \log p)$, a factor of $\log M / \log p$ faster than a multi-exponentiation whose powers have bit-lengths $\log p$. The verification of $\mathbf{e}$ costs $O(d/\log d)$ g.e. The aggregation cost is small.

6 Experiments

We implement the proposed RiseFL system in C/C++. The cryptographic primitives are based on libsodium[5] that implements the Ristretto group[6] on Curve25519 that supports 126-bit security. We compile it to a Python library using SWIG[7], and integrate the library into FedML[8] [26]. The implementation consists of 9K lines of code in C/C++ and 1.1K lines of code in Python.

6.1 Methodology

Experimental Setup. We conduct the micro-benchmark experiments on a single server equipped with Intel(R) Core(TM) i7-8550U CPU and 16GB of RAM and the federated learning tasks on eight servers with Intel(R) Xeon(R) W-2133 CPU, 64GB of RAM, and GeForce RTX 2080 Ti. The default security parameter is 126 bits. We set $\varepsilon = 2^{-128}$ to ensure the level of security of RiseFL matches the baselines. For the fixed-point integer representation, we use 16 bits to encode the floating-point values by default. As discussed in Section 5, we set $M = 2^{24}$ to make sure the rounding error of discrete normal samples is small.

Datasets. We use two real-world datasets, i.e., MNIST [17] and CIFAR10 [30], to run the FL tasks for measuring classification accuracy. The MNIST dataset consists of 70000 28 × 28 images in 10 classes. The CIFAR-10 dataset consists of 60000 32 × 32 color images in 10 classes. We also generate a synthetic dataset for micro-benchmarking the computational cost and communication cost.

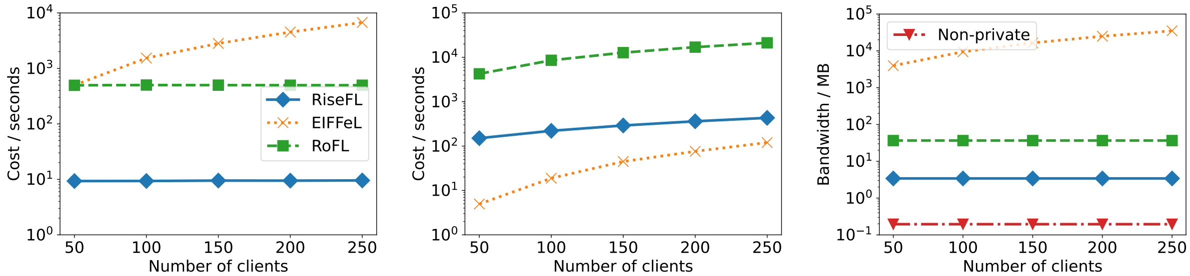
[5] https://doc.libsodium.org/
[6] https://ristretto.group/
[7] https://www.swig.org/
[8] https://github.com/FedML-AI/FedML

Models. We employ a CNN model for the MNIST dataset and ResNet-20 [27] for the CIFAR-10 dataset. The CNN model consists of four layers and 1.2M parameters. The ResNet-20 model consists of 20 layers and 270K parameters. For the micro-benchmark experiments on synthetic datasets, we use synthetic models with 1K, 10K, 100K, and 1M parameters for the evaluation.

Baselines. We compare RiseFL with two secure aggregation with verified inputs (SAVI) baselines, namely RoFL [9] and EIFFeL [45] using the same secure parameter, to evaluate the performance. Furthermore, we compare RiseFL with two non-private baselines for input integrity checking to evaluate the model accuracy. The following describes the baselines:

- *RoFL* [9] adopts the ElGammal commitment scheme and uses the strict checking zero-knowledge proof for integrity check. It does not guarantee Byzantine robustness.
- *EIFFeL* [45] employs the verifiable Shamir secret sharing (VSSS) scheme and secret-shared non-interactive proofs (SNIP) for SAVI, ensuring Byzantine robustness. We implement EIFFeL (see Appendix A.4) as it is not open-sourced.
- *NP-SC* is a non-private baseline with strict integrity checking. That is, the server checks each client's update and eliminates the update that is out of the L_2-norm bound.
- *NP-NC* is a non-private baseline without any checking on clients' updates. In this baseline, malicious clients can poison the aggregated models through malformed updates.

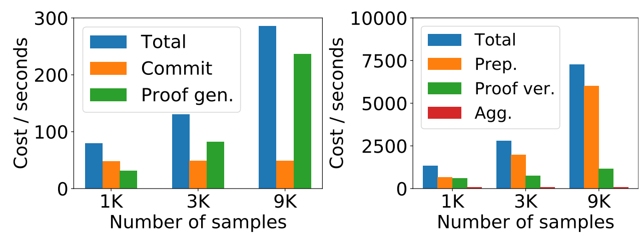
Metrics. We utilize three metrics to evaluate the performance of the RiseFL system.



(a) Client computational cost (b) Server computational cost (c) Communication cost per client

Figure 6: Cost comparison w.r.t. the number of clients

- *Computational Cost* refers to the computation time on each client and on the server, which measures the computational efficiency of the protocol.
- *Communication Cost per Client* refers to the size of messages transmitted between the server and each client, which measures the communication efficiency.
- *Model Accuracy* measures the ratio of correct predictions for the trained FL model, which is used to measure the effectiveness of the integrity check method in RiseFL.



(a) Client cost (b) Server cost

Figure 7: Cost v.s. the number of samples (d = 1M)

6.2 Micro-Benchmark Efficiency Evaluation

We first compare the cost of our RiseFL with EIFFeL and RoFL using micro-benchmark experiments. Unless otherwise specified, we set the number of clients to 100 and the maximum number of malicious clients to 10 in this set of experiments. We use 16 bits[9] for encoding floating-point values and run the experiments on one CPU thread on both the client side and the server side.

Effects of d. Tables 2 shows the cost comparison of RiseFL, EIFFeL and RoFL, w.r.t. the number of parameters d in the FL model, on the client computational cost, the server computational cost, and the communication cost per client. We set the number of samples $k = 1000$. We failed to run the experiments with d = 1M on EIFFeL and RoFL due to insufficient RAM, i.e., out of memory (OOM).

We observe that RiseFL is superior to the baselines when $d >> k$. For example, when d = 100K, the client cost of RiseFL is 53x smaller than RoFL and 164x smaller than EIFFeL. Compared to RoFL, the savings occur at the proof generation stage, which is because of our probabilistic check technique. The cost of EIFFeL is large as every client is responsible for computing a proof digest of every other client's proof in the proof verification stage, which dominates the cost. In comparison, the client-side proof verification cost of RiseFL is negligible since every client $\mathcal{C}_i$ only verifies the check string of the Shamir share of one value r_j from every other client $\mathcal{C}_j$. This is in line with the theoretical cost analysis in Table 1.

On the aspect of server cost, RiseFL is 39x faster than RoFL when d = 100K, due to our probabilistic check method. RiseFL incurs a higher server cost than EIFFeL because in EIFFeL, the load of proof verification is on the client side, and the dominating cost of the server is at the aggregation stage. In fact, the total server cost of RiseFL is 7 times smaller than the cost of EIFFeL of every client.

For communication cost, when d = 100K, RiseFL results in 16x more than transmitting the weight updates in clear text: the committed value of a 16-bit weight update is 256-bit long. The proof size is negligible because $k << d$. RoFL transmits about 10x more elements than RiseFL in the form of proofs of well-formedness and proofs of squares. The communication cost of EIFFeL is three orders of magnitude larger than RiseFL as it transmits the share of every bit of every coordinate of the weight update to every other client.

Effects of n. Figure 6 compares the computational cost and communication cost per client in terms of the number of clients. We vary the number of clients $n \in \{50, 100, 150, 200, 250\}$, and set the number of parameters d = 100K, and the maximum number of malicious clients $m = 0.1n$ in EIFFeL and RiseFL. Besides, we set the number of samples k = 1K of RiseFL in this experiment. From Figures 6a and 6c, we can observe that both the client computational cost and communication cost per client of RiseFL are

[9]The effect of bit-length on the computational cost of RiseFL is small, detailed in Appendix A.5.

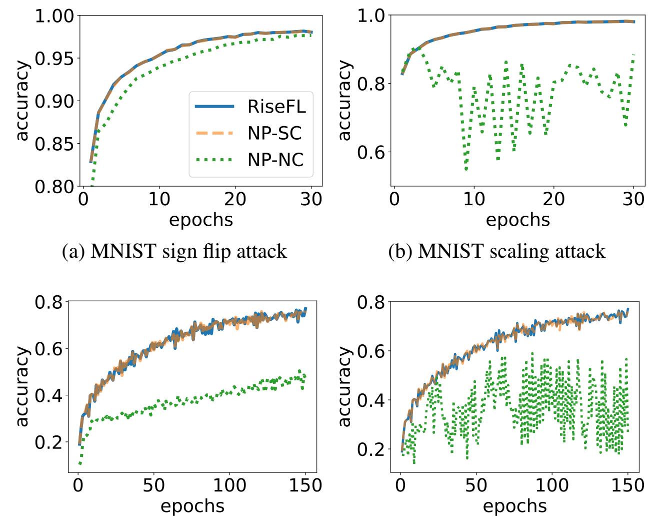
at least one order of magnitude lower than RoFL and EIFFeL, while the server cost of RiseFL is linear in n. In comparison, the cost of EIFFeL both on the client side and the server side increases quadratically in n. When n gets larger, the advantage of RiseFL is even larger compared to EIFFeL.

Effects of k. Figure 7 is the per-stage breakdown of RiseFL with $d = 1\text{M}$ by varying $k \in \{1\text{K}, 3\text{K}, 9\text{K}\}$. On the client side, proof generation is the only stage that scales with k. On the server side, as k gets larger, the preparation cost of computing $\mathbf{h}$ becomes dominant. The effects of k on the cost breakdown can be interpreted by Table 1. The terms that scale linearly with k are the $O(kd)$ f.a. term of the client's proof generation and the server's preparation costs. The linear-in-k terms become dominant as k becomes larger.

6.3 Robustness Evaluation

To evaluate the effectiveness of our probabilistic input integrity check method, we test the FL model accuracy of RiseFL on MNIST and CIFAR-10 against two commonly used attacks. The first is the sign flip attack [15], where each malicious client submits $-c \cdot \mathbf{u}$ as its model update and $c > 1$. The second is the scaling attack [4], where each malicious client submits $c \cdot \mathbf{u}$ as its model update and $c > 1$. In this set of experiments, we choose $c = 1.5$ for the sign flip attack and $c = 10$ for the scaling attack. We use 24 bits to encode the floating-point values in the updates, and set $n = 16$ and $m = 2$.

Figure 8 compares the training curves of RiseFL with two non-private baselines, NP-SC and NP-NC (see Section 6.1). There are two main observations. First, RiseFL achieves better accuracy than the no-checking baseline NP-NC. This is expected as the malicious clients can poison the aggregated models by invalid model updates when the server does not check the input integrity, leading to lower accuracy (see Figures 8a and 8c) or non-converging curves (see Figures 8b and 8d). Second, the training curves of RiseFL and the strict L_2 norm check baseline NP-SC are very close. This validates the effectiveness of RiseFL in identifying malformed updates and robust aggregation.



(a) MNIST sign flip attack
(b) MNIST scaling attack
(c) CIFAR10 sign flip attack
(d) CIFAR10 scaling attack

Figure 8: Comparison of training curves

7 Related Works

Secure Aggregation. To protect the client's input privacy in federated learning (FL), a number of studies have explored secure aggregation [3, 7, 28, 57], which enables the server to compute the aggregation of clients' model updates without knowing individual updates. A widely adopted approach [7] is to let each client use pairwise random values to mask the local update before uploading it to the server. The server can then securely cancel out the masks for correct aggregation. Nonetheless, these solutions do not guarantee input integrity, as malicious clients can submit arbitrary masked updates.

Robust Learning. Several works have been proposed for robust machine learning, including [1, 5, 11, 14, 33, 34, 41, 47, 48, 52, 54, 55]. However, some of these approaches, such as [14, 33, 47, 48], are designed for centralized training and require access to the training data, making them unsuitable for FL. On the other hand, solutions like [1, 5, 11, 34, 41, 52, 54, 55] specifically address the FL setting and focus on ensuring Byzantine resilient gradient aggregation. These solutions operate by identifying and eliminating client updates that deviate significantly from the majority of clients' updates, as they are likely to be malformed updates. However, it is worth noting that these approaches require the server to access the plaintext model updates, which compromises the client's input privacy.

Input Integrity Check with Secure Aggregation. There is Prio [13] that ensures both input privacy and input integrity with multiple non-collaborating servers. In contrast, our paper focuses on a single-server setting. Under a single-server setting, [9, 45] ensure both input privacy and input integrity. RoFL [9] utilizes homomorphic commitments [22] that are compatible with existing mask-based secure aggregation methods and adopt the zero-knowledge proof [8] to validate the clients' inputs. However, it does not support Byzantine-robust aggregation. EIFFeL [45] designs an approach based on verifiable secret sharing [46] and secret-shared non-interactive proofs (SNIP) [13] techniques, which tolerates Byzantine attacks. Nevertheless, the efficiency of these two solutions is quite low, especially when the number of model parameters is large. In contrast, in RiseFL, we propose a hybrid commitment scheme and design a probabilistic input integrity check method, providing support for Byzantine-robust aggregation and achieving significant efficiency improvements.

8 Conclusions

In this paper, we propose RiseFL, a robust and secure federated learning system that guarantees both input privacy and input integrity of the participating clients. We design a hybrid commitment scheme based on Pedersen commitment and verifiable Shamir secret sharing, and present a probabilistic L_2-norm integrity check method, which achieves a comparable security guarantee to state-of-the-art solutions while significantly reducing the computation and communication costs. The experimental results confirm the efficiency and effectiveness of our solution.

References

[1] Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *NeurIPS*, pages 4618–4628, 2018.

[2] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, pages 2938–2948, 2020.

[3] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In *CCS*, pages 1253–1269, 2020.

[4] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin B. Calo. Analyzing federated learning through an adversarial lens. In *ICML*, pages 634–643, 2019.

[5] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NeurIPS*, pages 119–129, 2017.

[6] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112, 1988.

[7] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, pages 1175–1191, 2017.

[8] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *S&P*, pages 315–334. IEEE, 2018.

[9] Lukas Burkhalter, Hidde Lycklama, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. Rofl: Attestable robustness for secure federated learning. *arXiv preprint arXiv:2107.03311*, 2021.

[10] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. *Technical Report/ETH Zurich, Department of Computer Science*, 260, 1997.

[11] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. In *NDSS*, 2021.

[12] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *CoRR*, abs/1712.05526, 2017.

[13] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[14] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *S&P*, pages 81–95, 2008.

[15] Georgios Damaskinos, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous byzantine machine learning (the case of sgd). In *ICML*, pages 1145–1154. PMLR, 2018.

[16] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136, 2001.

[17] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[18] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security*, pages 1605–1622, 2020.

[19] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–438. IEEE, 1987.

[20] Chong Fu, Xuhong Zhang, Shouling Ji, Jinyin Chen, Jingzheng Wu, Shanqing Guo, Jun Zhou, Alex X. Liu, and Ting Wang. Label inference attacks against vertical federated learning. In *USENIX Security Symposium*, pages 1397–1414, 2022.

[21] Clement Fung, Chris J. M. Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning. *CoRR*, abs/1808.04866, 2018.

[22] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[23] Shuhong Gao. A new algorithm for decoding reed-solomon codes. *Communications, information and network security*, pages 55–68, 2003.

[24] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques.* Cambridge University Press, 2001.

[25] Jamie Hayes and Olga Ohrimenko. Contamination attacks and mitigation in multi-party machine learning. In *NeurIPS*, pages 6604–6616, 2018.

[26] Chaoyang He, Songze Li, Jinhyun So, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[28] Peter Kairouz, Ziyu Liu, and Thomas Steinke. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *ICML*, pages 5201–5212, 2021.

[29] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, pages 1575–1590, 2020.

[30] Alex Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009.

[31] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[32] Junxu Liu, Jian Lou, Li Xiong, Jinfei Liu, and Xiaofeng Meng. Projected federated averaging with heterogeneous differential privacy. *PVLDB*, 15(4):828–840, 2021.

[33] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, pages 273–294, 2018.

[34] Xu Ma, Xiaoqian Sun, Yuduo Wu, Zheli Liu, Xiaofeng Chen, and Changyu Dong. Differentially private byzantine-robust federated learning. *IEEE Trans. Parallel Distributed Syst.*, 33(12):3690–3701, 2022.

[35] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, pages 1273–1282, 2017.

[36] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *S&P*, pages 691–706, 2019.

[37] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, 1978.

[38] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *S&P*, pages 739–753, 2019.

[39] Thien Duc Nguyen, Phillip Rieger, Huili Chen, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Shaza Zeitouni, Farinaz Koushanfar, Ahmad-Reza Sadeghi, and Thomas Schneider. FLAME: taming backdoors in federated learning. In *USENIX Security Symposium*, pages 1415–1432, 2022.

[40] Thuy Dung Nguyen, Tuan Nguyen, Phi Le Nguyen, Hieu H. Pham, Khoa Doan, and Kok-Seng Wong. Backdoor attacks and defenses in federated learning: Survey, challenges and future research directions. *CoRR*, abs/2303.02213, 2023.

[41] Xudong Pan, Mi Zhang, Duocai Wu, Qifan Xiao, Shouling Ji, and Min Yang. Justinian's gaavernor: Robust distributed learning with gradient aggregation agent. In *USENIX Security*, pages 1641–1658, 2020.

[42] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140. Springer, 2001.

[43] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

[44] C. Pomerance and S. Goldwasser. *Cryptology and Computational Number Theory*. AMS short course lecture notes. American Mathematical Society, 1990.

[45] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. In *CCS*, pages 2535–2549, 2022.

[46] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[47] Yanyao Shen and Sujay Sanghavi. Learning with bad training data via iterative trimmed loss minimization. In *ICML*, pages 5739–5748, 2019.

[48] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks. In *NeurIPS*, pages 3517–3529, 2017.

[49] Timothy Stevens, Christian Skalka, Christelle Vincent, John Ring, Samuel Clark, and Joseph P. Near. Efficient differentially private secure aggregation for federated learning via hardness of learning with errors. In *USENIX Security Symposium*, pages 1379–1395, 2022.

[50] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H Brendan McMahan. Can you really backdoor federated learning? *arXiv preprint arXiv:1911.07963*, 2019.

[51] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. DBA: distributed backdoor attacks against federated learning. In *ICLR*, 2020.

[52] Chang Xu, Yu Jia, Liehuang Zhu, Chuan Zhang, Guoxie Jin, and Kashif Sharif. TDFL: truth discovery based byzantine robust federated learning. *IEEE Trans. Parallel Distributed Syst.*, 33(12):4835–4848, 2022.

[53] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM TIST*, 10(2):12:1–12:19, 2019.

[54] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter L. Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *ICML*, pages 5636–5645, 2018.

[55] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter L. Bartlett. Defending against saddle point attack in byzantine-robust distributed learning. In *ICML*, pages 7074–7084, 2019.

[56] Hongxu Yin, Arun Mallya, Arash Vahdat, Jose M. Alvarez, Jan Kautz, and Pavlo Molchanov. See through gradients: Image batch recovery via gradinversion. In *CVPR*, pages 16337–16346, 2021.

[57] Yifeng Zheng, Shangqi Lai, Yi Liu, Xingliang Yuan, Xun Yi, and Cong Wang. Aggregation service for federated learning: An efficient, secure, and more resilient realization. *IEEE Trans. Dependable Secur. Comput.*, 20(2):988–1001, 2023.

[58] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In *NeurIPS*, pages 14747–14756, 2019.

Algorithm 5 $\mathsf{GenPrfSq}(g,h,\mathbf{y}_1,\mathbf{y}_2,\mathbf{x},\mathbf{r}_1,\mathbf{r}_2)$

Randomly sample $\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3 \in \mathbb{Z}_p^k$.
Compute $t_{1i} = g^{v_{1i}} h^{v_{2i}}$, $t_{2i} = y_{1i}^{v_{1i}} h^{v_{3i}}$ for $i \in [1,k]$.
Compute $c = H(g,h,\mathbf{y}_1,\mathbf{y}_2,\mathbf{t}_1,\mathbf{t}_2)$.
Compute $\mathbf{s}_1 = \mathbf{v}_1 - c\mathbf{x}_1$, $\mathbf{s}_2 = \mathbf{v}_2 - c\mathbf{r}_1$, and $\mathbf{s}_3 = \mathbf{v}_3 - c(\mathbf{r}_2 - \mathbf{r}_1 \circ \mathbf{x})$.
return $\pi = (\mathbf{t}_1,\mathbf{t}_2,\mathbf{s}_1,\mathbf{s}_2,\mathbf{s}_3)$.

Algorithm 6 $\mathsf{VerPrfSq}(g,h,\mathbf{y}_1,\mathbf{y}_2,\pi)$

Unravel $\pi = (\mathbf{t}_1,\mathbf{t}_2,\mathbf{s}_1,\mathbf{s}_2,\mathbf{s}_3)$.
Randomly sample $\alpha_i,\beta_i \in \mathbb{Z}_p$ for $i = 1,\dots,k$.
Compute $c = H(g,h,\mathbf{y}_1,\mathbf{y}_2,\mathbf{t}_1,\mathbf{t}_2)$.
return $g^{\sum_{i=1}^k \alpha_i s_{1i}} h^{\sum_{i=1}^k \alpha_i s_{2i} + \beta_i s_{3i}} \prod_{i=1}^k y_{1i}^{\alpha_i c + \beta_i s_{1i}} y_{2i}^{c\beta_i} == \prod_{i=1}^k t_{1i}^{\alpha_i} t_{2i}^{\beta_i}$.

Algorithm 7 $\mathsf{GenPrfWf}(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},r,\mathbf{v}^*,\mathbf{s})$

Randomly Sample $w, x_0,\dots,x_k,x_1',\dots,x_k' \in \mathbb{Z}_p$.
Compute $u = g^w$, $t_i = g^{x_i} h_i^w$ $(\forall i \in [0,k])$, $t_i^* = g^{x_i} q^{x_i'}$ $(\forall i \in [1,k])$.
Compute $c = H(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},u,\mathbf{t},\mathbf{t}^*)$.
Compute $y = w - cr$, $y_i = x_i - cv_i$ $(\forall i \in [0,k])$, $y_i' = x_i' - cs_i$ $(\forall i \in [1,k])$, where $\mathbf{v}^* = (v_0,\dots,v_k)$
return $\pi = (u,\mathbf{t},\mathbf{t}^*,y,\mathbf{y},\mathbf{y}')$.

Algorithm 8 $\mathsf{VerPrfWf}(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},\pi)$

Unravel $\pi = (u,\mathbf{t},\mathbf{t}^*,y,\mathbf{y},\mathbf{y}')$.
Compute $c = H(g,q,\mathbf{h},z,\mathbf{e},\mathbf{o},u,\mathbf{t},\mathbf{t}^*)$.
Randomly sample $\alpha, \beta_i (i \in [0,k]), \gamma_i (i \in [1,k]) \in \mathbb{Z}_p$.
return $u^{\alpha} \prod_{i=0}^k t_i^{\beta_i} \prod_{i=1}^k t_i^{*\gamma_i} == g^{\alpha w + \sum_{i=0}^k \beta_i y_i + \sum_{i=1}^k \gamma_i y_i} z^{\alpha c} \prod_{i=0}^k (h_i^{\beta_i y} e_i^{\beta_i c}) q^{\sum_{i=1}^k \gamma_i y_i'} \prod_{i=1}^k o_i^{c\gamma_i}$.

A Appendix

A.1 Preliminaries Extension

Σ-protocol for proof of squares $((\mathbf{x},\mathbf{r}_1,\mathbf{r}_2),(\mathbf{y}_1,\mathbf{y}_2))$. Denote $g,h \in \mathbb{G}$ the independent group elements. Given secrets $\mathbf{x},\mathbf{r}_1,\mathbf{r}_2 \in \mathbb{Z}_p^k$ and commitments $(\mathbf{y}_1,\mathbf{y}_2) = (g^{\mathbf{x}} h^{\mathbf{r}_1}, g^{\mathbf{x}^2} h^{\mathbf{r}_2})$, i.e. $y_{1i} = g^{x_i} h^{r_{1i}}$ and $y_{2i} = g^{x_i^2} h^{r_{2i}}$ for $i \in [1,k]$, the prover uses the function $\mathsf{GenPrfSq}()$ in Algorithm 5 to generate a proof π that the secret in y_{2i} is the square of the secret in y_{1i} for every i. The verifies uses the function $\mathsf{VerPrfSq}()$ in Algorithm 6 to verify this proof based on $(\mathbf{y}_1,\mathbf{y}_2)$. In Algorithm 6, the random numbers α_i,β_i are used for batch verification of multiple equalities: $g^{s_{1i}} h^{s_{2i}} y_{1i}^c == t_{1i}$, $h^{s_{3i}} y_{1i}^{s_{1i}} y_{2i}^c == t_{2i}$ for $i \in [1,k]$. Batch-verifying these equalities saves cost by a factor of $O(\log(k))$.

Σ-protocol for proof of relation $((r,\mathbf{v}^*,\mathbf{s}),(z,\mathbf{e},\mathbf{o}))$. Given independent group elements $g,q,h_0,\dots,h_k \in \mathbb{G}$, the Σ-protocol to prove and verify that $z = g^r$, $e_i = g^{v_i} h_i^r$ $(i \in [0,k])$, $o_i = g^{v_i} q^{s_i}$ $(i \in [1,k])$, where $v^* = (v_0,\dots,v_k)$, is the pair of functions $\mathsf{GenPrfWf}()$ and $\mathsf{VerPrfWf}()$. The function $\mathsf{GenPrfWf}()$ in Algorithm 7 generates a proof π that $(z,\mathbf{e},\mathbf{o})$ is of the form $(g^r, g^{\mathbf{v}^*}\mathbf{h}^r, g^{\mathbf{v}} q^{\mathbf{s}})$, where $\mathbf{v} = (v_1,\dots,v_k)$. The function $\mathsf{VerPrfWf}()$ in Algorithm 8 verifies this proof. Again, we use batch verification to verify multiple equalities: $u == g^w z^c$, $t_i == g^{y_i} h_i^y e_i^c$ $(i \in [0,k])$, $t_i^* == g^{y_i} q^{y_i'} o_i^c$ $(i \in [1,k])$.

A.2 Chi-square Distribution of Sampling

A.2.1 Proof of Lemma 1

Proof of Lemma 1. Since $\mathbf{a}_t$ follows $\mathcal{N}(\mathbf{0},\mathbf{I}_d)$, its projection on the direction of $\mathbf{u}$, $\langle \mathbf{a}_t, \frac{\mathbf{u}}{||\mathbf{u}||}\rangle$, follows $\mathcal{N}(0,1)$. Therefore, the sum of squares of these inner products for $t \in [1,k]$,

$$\frac{1}{||\mathbf{u}||_2^2} \sum_{t=1}^k \langle \mathbf{a}_t,\mathbf{u}\rangle^2,$$

follows χ_k^2. □

A.2.2 Proof of Lemma 2

Proof of Lemma 2. We have $|b_{tj} - a_{tj}| \le 1/2$ for all t,j. So $||\mathbf{a}_t - \mathbf{b}_t||_2 \le \sqrt{d}/2$. For every t, we have

$$\begin{aligned}\langle \mathbf{a}_t,\mathbf{u}\rangle^2 - \langle \mathbf{b}_t,\mathbf{u}\rangle^2 &= \langle \mathbf{a}_t - \mathbf{b}_t,\mathbf{u}\rangle^2 + 2\langle \mathbf{b}_t,\mathbf{u}\rangle\langle \mathbf{a}_t - \mathbf{b}_t,\mathbf{u}\rangle \\ &\le ||\mathbf{a}_t - \mathbf{b}_t||_2^2 ||\mathbf{u}||_2^2 + 2||\mathbf{a}_t - \mathbf{b}_t|| \cdot ||\mathbf{u}|| \cdot |\langle \mathbf{b}_t,\mathbf{u}\rangle| \\ &\le \frac{1}{4} dB^2 + \sqrt{d} \cdot B \cdot |\langle \mathbf{b}_t,\mathbf{u}\rangle|.\end{aligned}$$

Summing up and using $\sum_{t=1}^k |\langle \mathbf{b}_t,\mathbf{u}\rangle| \le \sqrt{k \sum_{t=1}^k \langle \mathbf{b}_t,\mathbf{u}\rangle^2}$, we have

$$\sum_{t=1}^k \left(\langle \mathbf{a}_t,\mathbf{u}\rangle^2 - \langle \mathbf{b}_t,\mathbf{u}\rangle^2\right) \le \frac{1}{4} k dB^2 + \sqrt{d} \cdot B \sqrt{k \sum_{t=1}^k \langle \mathbf{b}_t,\mathbf{u}\rangle^2}$$

Adding this to the assumption that $\sum_{t=1}^k \langle \mathbf{b}_t,\mathbf{u}\rangle^2 \le B^2 M^2 r$ yields the desired inequality. □

A.2.3 Proof of Lemma 3

Proof of Lemma 3. Continuing with the idea of the proof of Lemma 2, we have

$$\begin{aligned}\langle \mathbf{a}_t,\mathbf{u}\rangle^2 - \langle \mathbf{b}_t,\mathbf{u}\rangle^2 &= \langle \mathbf{a}_t - \mathbf{b}_t,\mathbf{u}\rangle^2 + 2\langle \mathbf{b}_t,\mathbf{u}\rangle\langle \mathbf{a}_t - \mathbf{b}_t,\mathbf{u}\rangle \\ &\ge -2||\mathbf{a}_t - \mathbf{b}_t|| \cdot ||\mathbf{u}|| \cdot |\langle \mathbf{b}_t,\mathbf{u}\rangle| \\ &\ge -\sqrt{d} \cdot B \cdot |\langle \mathbf{b}_t,\mathbf{u}\rangle|.\end{aligned}$$

Summing up, we have

$$\sum_{t=1}^k \left(\langle \mathbf{a}_t,\mathbf{u}\rangle^2 - \langle \mathbf{b}_t,\mathbf{u}\rangle^2\right) \ge -\sqrt{d} \cdot B \sqrt{k \sum_{t=1}^k \langle \mathbf{b}_t,\mathbf{u}\rangle^2}.$$

After moving $\sum_{t=1}^{k}\langle\langle \mathbf{b}_t, \mathbf{u}\rangle^2$ to the right and completing the square, we get

$$\begin{aligned}\sqrt{\sum_{t=1}^{k}\langle \mathbf{b}_t, \mathbf{u}\rangle^2} &\le \sqrt{\sum_{t=1}^{k}\langle \mathbf{a}_t, \mathbf{u}\rangle^2 + \left(\frac{B\sqrt{kd}}{2}\right)^2} + \frac{B\sqrt{kd}}{2} \\ &\le \sqrt{\sum_{t=1}^{k}\langle \mathbf{a}_t, \mathbf{u}\rangle^2} + B\sqrt{kd} \\ &\le \sqrt{B_0} + B\sqrt{kd} \\ &= BM\left(\sqrt{r_{k,\varepsilon}} + \frac{3\sqrt{kd}}{2M}\right).\end{aligned}$$

After squaring both sides and dividing by $||\mathbf{u}||_2^2$, we obtain the desired inequality. □

A.3 The Complete Security Proof

A.3.1 Proof of Lemma 4

Proof. We have $||\mathbf{u}_i||_2 \le B$. By Lemma 1, the probability that

$$\sum_{t=1}^{k}\langle \mathbf{b}_t, \mathbf{u}_i\rangle^2 \le B^2M^2\gamma_{k,\varepsilon}$$

is at least $1-\varepsilon$. By Lemma 2, the probability that

$$\sum_{t=1}^{k}\langle \mathbf{a}_t, \mathbf{u}_i\rangle^2 \le B_0$$

is at least $1-\varepsilon$. If Eqn A.3.1 holds, $\mathcal{C}_i$ can produce a proof which passes the integrity check. □

A.3.2 Proof of Lemma 5

Proof of Lemma 5. The function $\mathsf{VerPrfWf}$ and the large sampling space $\mathbb{Z}_p$ on each coordinate ensures that $\mathcal{C}_i$ must be able to efficiently respond to any $\mathbf{a}_0$ by producing v_0 and r that satisfies $g^{v_0}h_0^r = e_0$. The function VerCrt ensures that $e_0 = \prod_{l=1}^{d} y_{il}^{a_{0l}}$. That is,

$$g^{v_0}h_0^r = \prod_{l=1}^{d} y_{il}^{a_{0l}}. \tag{9}$$

If we change $\mathbf{a}_0$ to $\mathbf{a}_0'$, $\mathcal{C}_i$ produces v_0' and r' that satisfies $g^{v_0'}(h_0')^{r} = \prod_{l=1}^{d} y_{il}^{a_{0l}'}$. Therefore,

$$g^{v_0'-v_0}\prod_{l=1}^{d} w_l^{a_{0l}'r'-a_{0l}r} = \prod_{l=1}^{d} y_{il}^{a_{0l}'-a_{0l}}.$$

For each l, by setting $a_{0j}' = a_{0j} + \delta_{lj}$ where $\delta_{lj} = 1$ if $l = j$, $\delta_{lj} = 0$ if $l \ne j$, we get that each y_{il} can be expressed into the form

$$y_{il} = g^{\alpha_l}\prod_{j=1}^{d} w_j^{\beta_{lj}}.$$

We would like to have $\beta_{lj} = 0$ whenever $l \ne j$. Suppose not and without loss of generality, $\beta_{12} \ne 0$. Equation 9 becomes

$$g^{v_0}\prod_l w_l^{ra_{0l}} = g^{\sum_l a_{0l}\alpha_l}\prod_l w_l^{\sum_j a_{0j}\beta_{jl}}.$$

If we change $\mathbf{a}_0$ to $\mathbf{a}_0'$ where $a_{01}' = a_{01}+1$, $a_{0l}' = a_{0l}$ for $l \ge 2$ and $\mathcal{C}_i$ produces v_0' and r', we get

$$g^{v_0'}\prod_l w_l^{r'a_{0l}'} = g^{\sum_l a_{0l}'\alpha_l}\prod_l w_l^{\sum_j a_{0j}'\beta_{jl}}.$$

Dividing these two inequalities, we get

$$g^{v_0'-v_0}\prod_l w_l^{r'a_{0l}'-ra_{0l}} = g^{\alpha_1}\prod_l w_l^{\beta_{1l}}.$$

Because $g, w_1, \ldots, w_d$ are independent, we must have $v_0' - v_0 = \alpha_1$, $r'a_{0l}' - ra_{0l} = \beta_{1l}$. So $(r'-r)a_{01} + r' = \beta_{11}$, $(r'-r)a_{0l} = \beta_{1l}$ for $l \ge 2$. If $r' \ne r$, then we must have $a_{02}^{-1}a_{03} = \beta_{12}^{-1}\beta_{13}$. So whenever $a_{02}^{-1}a_{03} \ne \beta_{12}^{-1}\beta_{13}$, we must have $r' = r$, and then $r' = \beta_{11}$, so

$$r'a_{0l}' = \beta_{11}a_{0l}' = \sum_j a_{0j}'\beta_{jl}.$$

for any l. So $\beta_{11} = \beta_{ll}$ and $\beta_{jl} = 0$ whenever $j \ne l$. This is a contradiction.

We have concluded that $\beta_{lj} = 0$ whenever $l \ne j$. So

$$g^{v_0}\prod_l w_l^{ra_{0l}} = g^{\sum_l a_{0l}\alpha_l}\prod_l w_l^{a_{0l}\beta_{ll}}.$$

Therefore, $v_0 = \sum_l a_{0l}\alpha_l$ and $r = \beta_{ll}$. This ensures that client $\mathcal{C}_i$'s commitment must satisfy $y_{il} = g^{\alpha_l}w_l^r$, which is exactly the required form in the protocol.

So far, we have shown well-formedness of $\mathbf{y}_i$. To pass the integrity check, client $\mathcal{C}_i$ must submit correct computations of e_t, submit o_t, o_t' of the correct form, and submit boundedness proofs for each inner product and final sum. So $\mathbf{u}_i$ must satisfy equation A.3.1. □

A.3.3 Proof of Lemma 6

Proof. By Lemma 3, the probability that Eqn A.3.1 holds is at most $\Pr_{x\sim\chi_k^2}[x < \frac{1}{D(\mathbf{u}_i)^2}\left(\sqrt{\gamma_{k,\varepsilon}} + \frac{3\sqrt{kd}}{2M}\right)^2]$. If Eqn A.3.1 does not hold, by Lemma 5, the probability that $\mathcal{C}_i$ produces a valid proof is $\mathrm{negl}(\kappa)$. The sum of these two probabilities is $F_{k,\varepsilon,d,M}(D(\mathbf{u}_i))$. □

A.3.4 Proof of Theorem 1

Proof. Input integrity is already proved in Lemma 7. We now prove input privacy. Given an adversary $\mathcal{A}$ that consists of the malicious server and the malicious clients $\mathcal{C}_M$ attacking the real interaction, we define a simulator $\mathcal{S}$ as required.

$\mathcal{S}$ is defined by modifying $\mathcal{A}$ at each step when the server or the malicious clients $\mathcal{C}_M$ read inputs from one of the honest

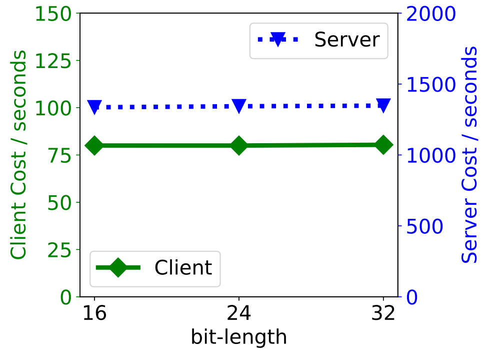


Figure 9: Cost comparison w.r.t. bit-length

clients $\mathcal{C}_H$. In the ideal functionality $\mathcal{F}$, the honest clients $\mathcal{C}_{\mathcal{H}}$ hold values $\{\mathbf{u}_i'\}_{\mathcal{C}_i \in \mathcal{C}_H}$ that is randomly picked from

$$\{\{\mathbf{u}_i''\}_{\mathcal{C}_i \in \mathcal{C}_H} : \sum_{\mathcal{C}_i \in \mathcal{C}_H} \mathbf{u}_i'' = \mathcal{U}_{\mathcal{H}}, \forall i \; \|\mathbf{u}_i''\| \le B\}.$$

In $\mathcal{S}$, the client $\mathcal{C}_i \in \mathcal{C}_H$, uses $\mathbf{u}_i'$ and random value r_i' as its weight update to make commitments. After receiving the samples $\mathbf{A}$ from the server, the client $\mathcal{C}_i \in \mathcal{C}_H$, sends a proof to the server if Eqn A.3.1 holds, and aborts if Eqn A.3.1 does not hold. By Lemma 4, the probability that one of the clients in $\mathcal{C}_H$ fails in $\mathcal{A}$ or $\mathcal{S}$ is $\text{negl}(\kappa)$. At the aggregation step, in both $\mathcal{A}$ and $\mathcal{S}$, the property of Shamir's sharing ensures that the server can only infer the sum of the secrets $\sum_{\mathcal{C}_i \in \mathcal{C}_H} r_i$, $\sum_{\mathcal{C}_i \in \mathcal{C}_H} r_i'$ respectively. The only information that the server can infer from this sum is $\mathcal{U}_H$.

Therefore, if Eqn A.3.1 holds for all $\mathcal{C}_i \in \mathcal{C}_H$ in both $\mathcal{A}$ and $\mathcal{S}$, the colluding party of the server and malicious clients $\mathcal{C}_M$ cannot infer anything from the proofs generated by $\mathcal{C}_H$ except $\mathcal{U}_H$, which means that

$$|\Pr[\text{Real}_{\Pi,\mathcal{A}}(\{\mathbf{u}_{\mathcal{C}_H}\}) = 1] - \Pr[\text{Ideal}_{\mathcal{F},\mathcal{S}}(\mathcal{U}_H) = 1]| \le \text{negl}(\kappa).$$

This inequality still holds after counting in the probability $\text{negl}(\kappa)$ that one of the clients in $\mathcal{C}_H$ fails the check. □

A.4 The Implementation Detail of EIFFeL

Since EIFFeL is not open-sourced, we implement it from scratch for a fair comparison. Two differences exist between our EIFFeL experiments and those in the original paper.

First, we add bound checks for every coordinate of the model updates, use the information-secure VSSS and the multiplicative homogeneity of Shamir's share to compute shares of the sum of squares, and use the batch checking to verify the check strings of VSSS, as discussed in Section 5.

Second, we use $3m+1$ shares, instead of n shares, to perform robust reconstruction [23] that tolerates m errors. This saves the cost of robust reconstruction by a factor of $n^2/(3m+1)^2$ compared to their original implementation.

A.5 Effects of bit-length of weight updates

We compare the effect of the integer bit-length that encodes clients' model updates on the client computational time and server computational time. We fix the number of parameters $d = 1\text{M}$, the number of samples $k = 1\text{K}$, the number of clients $n = 100$, the maximum number of malicious clients $m = 10$, and vary bit-length in $\{16, 24, 32\}$. Figure 9 shows the experimental results. We can observe that the effect of the bit-length on the cost is small.