Auditable Federated Learning With Byzantine Robustness

Yihuai Liang, Yan Li, and Byeong-Seok Shin, *Member, IEEE*

Abstract*— Machine learning (ML) has led to disruptive innovations in many fields, such as medical diagnoses. A key enabler for ML is large training data, but existing data, such as medical data, are not fully exploited by ML because of data silos and privacy concerns. Federated learning (FL) is a promising distributed learning paradigm to address this problem. On the other hand, existing FL approaches are vulnerable to poisoning attacks or privacy leakage from a malicious aggregator or client. This article proposes an auditable FL scheme with Byzantine robustness against the aggregator and client: The aggregator is malicious but available, and the client could perform poisoning attacks. First, the Pedersen commitment scheme (PCS) for homomorphic encryption was applied to preserve privacy and for commitments to the FL process to achieve auditability. The auditability enables clients to verify the correctness and consistency of the entire FL process and to identify parties that misbehave. Second, an efficient technique of divide and conquer was designed based on PCS to allow parties to cooperate and securely aggregate gradients to defend against poisoning attacks. This technique enables clients to share no common secret key and cooperate to decrypt ciphertext, guaranteeing a client's privacy even if some other clients are corrupted by adversaries. This technique was optimized to tolerate the dropout of clients. This article reports a formal analysis concerning privacy, efficiency, and auditability against malicious participants. Extensive experiments on various benchmark datasets show that the scheme is robust with high model accuracy against poisoning attacks.

***Index Terms*— Auditability, Byzantine robustness, federated learning (FL), poisoning attack, privacy preservation.**

Nomenclature

C	Set of clients.
m	Number of receivers.
$\lvert \cdot \rvert$	Number of elements in a set.
n	Dimension number of a gradient.
$G^{(i,)}$	Gradient vector of the ith client (not been split).
$G^{(i,j)}$	Split gradient of ith client sent to the jth receiver.
$g^{(i,)}$	Gradient value of the ith client (not been split).
$\bar{g}_k$	Baseline gradient in the kth dimension.
w	Client's weight for aggregation of the global model.
$[\![\cdot]\!]$	Homomorphic encryption via PCS.
$[a, b]$	All integers from a to b, including a and b.

Manuscript received 27 January 2023; revised 11 March 2023; accepted 3 April 2023. This work was supported in part by the Korea Institute for Advancement of Technology (KIAT) Grant funded by the Korean Government (MOTIE) under Grant P0017157 (World Class Plus Project) and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government (MSIT) [Artificial Intelligence Convergence Innovation Human Resources Development (Inha University)] under Grant RS-2022-00155915. *(Corresponding author: Byeong-Seok Shin.)*

The authors are with the Department of Electrical and Computer Engineering, Inha University, Incheon 22212, South Korea (e-mail: bsshin@inha.ac.kr).

Digital Object Identifier 10.1109/TCSS.2023.3266019

I. Introduction

MACHINE learning (ML) techniques have led to disruptive innovations in many fields, such as pathology, radiology, and genomics. A modern ML model, such as a medical diagnosis model, can contain millions of parameters and needs to be learned from sufficiently large data to achieve clinical-grade accuracy. On the other hand, data silos hinder ML from fully exploiting the data for building accurate and robust statistical models. Take health data as an example. Health data are difficult to obtain because they are highly sensitive, and their use is strictly regulated. In addition, collecting and maintaining a high-quality dataset incurs considerable time and expense. Such data may have high business value, making it unlikely to be shared freely. In addition to medical data, many other data, such as customer data of a bank and business data of a company, also have privacy concerns. Federated learning (FL) [1] is a promising distributed learning paradigm to address this problem. It trains an ML model across multiple clients who hold local data samples without exchanging them.

Although a client in FL submits gradients or model parameters to an aggregation server, previous studies [2], [3] have shown that adversaries can still infer privacy through the submitted data. Privacy-preserving FL (PPFL) [4], [5] has been studied widely by academia and industry. Homomorphic encryption, e.g., Paillier [6], is a common technique for encrypting gradients and still enables gradient aggregation over the ciphertext. Differential privacy [7] is another technique to mask the gradients before submitting them to the server. On the other hand, PPFL requires parties to be curious but honest. Nevertheless, it falls far short of the need to achieve robust FL.

Robust FL urgently needs to resist poisoning attacks [8] because, with malicious clients who perform poisoning attacks, FL is vulnerable [9], [10]. A single Byzantine client can significantly affect the trained model [11]. The privacy-preserving mechanism in PPFL increases the challenge of resisting poison attacks because poisonous gradients are difficult to observe or detect in ciphertext, which provides concealment for poisonous updates. On the other hand, poisoning attack defense strategies [11], [12] that have to access the local gradients could leak information. This contradiction between privacy preservation and poison attack detection makes it challenging to achieve secure FL.

In addition to poisoning attacks from malicious clients, robust FL requires a secure and trustworthy process of gradient aggregation against malicious aggregators. The aggregator could leak or tamper the gradients. It could deliberately skip aggregating a target client's gradient. In addition, it could even collude with a client to perform poisoning attacks without being effectively detected. The FL work [13] considers Byzantine clients with privacy preservation but assumes that the aggregator does not collude with clients. The recent FL works [14], [15], [16], which attempt to defend against poisoning attacks while preserving privacy, use two curious-but-honest servers and assume that the servers do not collude with one another or clients. Nevertheless, *who should be trusted to manage the servers?* In practice, the servers could be malicious, so such an assumption remains a potential security risk. Some approaches [17], [18] use blockchain as the aggregator to remove the reliance on such trusted authorities. However, public blockchain systems have limited performance concerning on-chain storage and smart contract computation. Using blockchain to collect gradients and aggregate gradients introduces new scalability issues. In addition, the transparency of blockchain may cause privacy leakage of gradient data. Although some blockchain-based FL approaches [19], [20], [21] address the privacy problem, they do not consider poison attacks against malicious clients. In summary, the existing work considers either malicious clients or Byzantine aggregators, not both.

This article proposes an auditable Byzantine-robustness FL (ABFL) scheme against the server and clients: The server is malicious but available, and clients could perform poisoning attacks. The server is curious about the client's privacy, can modify records, and can collude with clients. The clients can collude with each other or perform poisoning attacks. First, this scheme preserves client privacy via the homomorphic encryption of the Pedersen commitment scheme (PCS) [22]. Based on the PCS, this study designed a technique of divide and conquer, which enabled clients to share no common secret key for the homomorphic encryption, but can cooperate to decrypt the ciphertext. This technique provides the advantage that a client's privacy is secure even if some other clients are corrupted by adversaries. In the literature, the threshold Paillier algorithm [23], which is a trapdoor discrete logarithm scheme, achieves (t, ℓ)-threshold decryption: any subset of $t+1$ out of ℓ entities can decrypt a ciphertext. However, this Paillier algorithm has low performance, which is evaluated and presented in Section V-B. A variant of the ElGamal encryption scheme [24], which is a homomorphic cryptosystem, also achieved threshold decryption, but this ElGamal scheme has no trapdoor to recover the plaintext.

Second, this scheme offers auditability to guarantee the correctness of the entire FL process. The clients can verify if the result of each FL process is consistent with all clients' local gradients and preset agreements. Any misbehavior by clients or an aggregation server can be detected and identified. Such auditability should be a critical property in an FL system that involves multiple parties. The main idea is to commit the results of the FL process to the blockchain and to enable each participant to reperform the computation of the server without breaking client privacy. It is infeasible for a malicious party to forge the results because the PCS is computationally binding. Any inconsistent result can be detected based on the commitments on the blockchain. Third, an approach to establish an auditable and distributed root of trust was designed to eliminate the effect of malicious gradients. Clients are required to prepare small and clean root data to produce baseline gradients for secure gradient aggregation. The experimental results demonstrated the robustness of the design of the distributed root of trust, including when some clients do not exactly follow the agreement to prepare their local root data. The main contributions of this work are summarized as follows.

1) An ABFL scheme against malicious aggregation servers and clients is proposed.
2) A highly efficient technique of divide and conquer based on PCS is designed to allow parties to cooperate to decrypt homomorphic encryption securely.
3) A trusted global model is designed by removing malicious gradients to resist poisoning attacks.

The remainder of this article is organized as follows. Section II presents the system model and threat model. Section III describes the details of the scheme. Sections IV and V give the security and performance analysis, respectively. Section VI reviews related studies.

II. System Model

Fig. 1 presents the system architecture, which consists of three entities: clients, aggregation server, and public blockchain. Clients are data owners who want to train an ML model jointly using their local datasets. The aggregation server is a centralized aggregator, which is responsible for receiving gradients from the clients and calculating gradient aggregation. The server can be set up and managed by one of the clients. The public blockchain is used as a decentralized bulletin board to store commitments. The system focuses on a cross-silo setting [25], in which clients are a small number of organizations, such as financial and medical companies, with reliable communications and abundant computing resources.

A. Threat Model

A client may collude with others or the aggregation server. On the other hand, there are more than half of honest clients, who aim to benefit from the global model and upload the true gradients trained on their local datasets. Nevertheless, malicious clients are curious about other clients' sensitive information. They may make the system parameters and their local data public and perform poisoning attacks, such as uploading elaborately malicious gradients for backdoor attacks or submitting arbitrary gradients to reduce the accuracy of the global model. This study assumed that all clients were available during training. This assumption is reasonable in the cross-silo setting. The aggregation server could be malicious but available. It could infer clients' privacy, collude with some clients, and tamper with the intermediate results. Data stored on the server are available for all clients. The blockchain was assumed to be secure regarding data integrity.

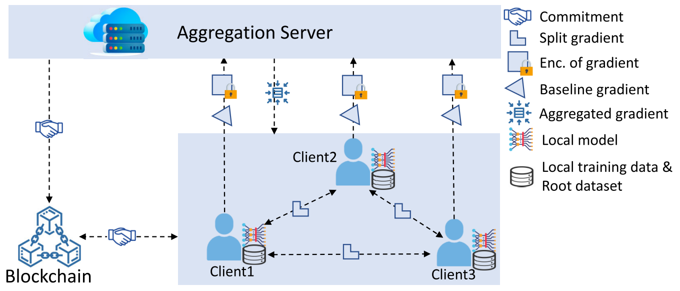

Fig. 1. Proposed system architecture for FL. Each client commits its gradient to the blockchain, splits the gradient, and shares the split gradients with other clients. The aggregation server receives encrypted gradients, securely aggregates gradients over homomorphic encryption, and commits intermediate results to the blockchain. A client downloads the aggregated gradient from the server and starts a new training iteration. After training, a client audits all FL processes based on the commitments.

III. Design of the Proposed Scheme

This section first describes an overview of the proposed approach and explains the concrete construction of the proposal. The main notations are presented in Nomenclature. The background knowledge on FL and PCS can be found in Appendixes A and B. To clarify the presentation, "receiver" was used to denote a client who receives other clients' split gradients, and "sender" denotes a client who sends its split gradients to other clients. A client can be both a receiver and a sender.

A. Overview of Our Proposed Approach

Fig. 2 presents a technical overview of the proposal. First, the ith client trains the local model with the local dataset and obtains the gradient $G^{(i,)}$. The client then normalizes the gradient and splits the gradient randomly such that the dimensionwise addition of the split gradients equals $G^{(i,)}$. $G^{(i,)}$ is encrypted via PCS before sending to the aggregation server, and the split gradients in the clear are shared with various clients. After obtaining all client's encrypted gradients, the server calculates the secure cosine similarity between the baseline gradient and each client's gradient. The server then opens (i.e., decrypts) the homomorphic encryption of the cosine similarity values using the opening values produced by the clients via the split gradients. The server then performs Byzantine-tolerance gradient aggregation based on those decrypted similarity values as weights to eliminate the malicious gradients. New opening values are obtained from the clients to open the aggregated gradient.

The PCS is used for homomorphic encryption to preserve the privacy of gradients and enable linear computation over the ciphertext. The linear computation includes weight calculation and gradient aggregation. Because the baseline gradient is in plaintext at the server, an inner product for the cosine similarity between the baseline gradient and a client's gradient can be calculated by scalar multiplication and additive homomorphic encryption of the PCS. Similarly, the linear computation can be conducted for the weighted gradient aggregation. However, PCS does not have a common private key among clients. Opening the ciphertext of the weights and the aggregated gradient requires the clients to produce opening values by the split gradients.

The PCS and blockchain were used to achieve auditability. PCS is a commitment scheme with homomorphic encryption. To audit the correctness of the server's computation, any party in this system can reperform the computation of the server without knowing the secret information. The correctness means all processing results of the FL training, including the weights, the aggregated gradient, and the opening values, are consistent with the commitments of all clients' gradients. For data integrity, blockchain, which has properties of immutability and traceability, was used to store the commitments of those results. Blockchain guarantees that the result of each training process is undeniable and cannot be altered. Hence, a party can audit the training process after the FL and effectively identify the entity that misbehaved.

The baseline gradient $\bar{G}$ is produced by clients with small and clean root datasets consistent with the settings in FLTrust [26]. Specifically, in addition to computing a local gradient using the local samples, a client also computes a local baseline gradient using a local root dataset. Unlike FLTrust, each client is required to secretly prepare a local root dataset and commit it to the blockchain. At each training iteration, each client computes a local baseline gradient using the local root dataset, commits the local baseline gradient on the blockchain, and uploads this gradient in plaintext to the server. The server then calculates the average of those local baseline gradients to obtain the final baseline gradient $\bar{G}$. If the cosine similarity between the final baseline gradient and a client's local gradient is less than zero, the client is malicious and will contribute negatively to the aggregation. Therefore, the server chooses to discard the gradient of this client at the current training iteration. The server then uses the similarities as weights to aggregate the gradients by computing a weighted average. After training, each client needs to reveal the root dataset such that other clients can audit whether a client appropriately chooses and uses the root dataset to generate the local baseline gradient according to a preset agreement.

B. Construction of ABFL

The parameters of the Pedersen commitment are publicly generated and shared with all parties. At the beginning of the protocol, the server initializes the parameters of the global model randomly. Those parameters and the hyperparameters of the model are shared with all clients.

1) Local Computing: Algorithm 1 presents the process of a client's local computation. In the tth iteration, each client uses its local dataset D and the global model W_{t-1} to calculate the local gradient in the following:

$$G_t \leftarrow \nabla L(D, W_{t-1}) \tag{1}$$

where L denotes the loss function and ∇ denotes the derivation operation. Similarly, the client uses the local root dataset to calculate the local baseline gradient. Each client commits their local root dataset before training. This dataset is revealed at the end of the FL.

The local gradients need to be normalized before encryption because a cosine similarity strategy is used to aggregate gradients. This normalization can allow the server to apply addition and scalar multiplication directly to the encryption without needing to deal with the magnitudes of the various

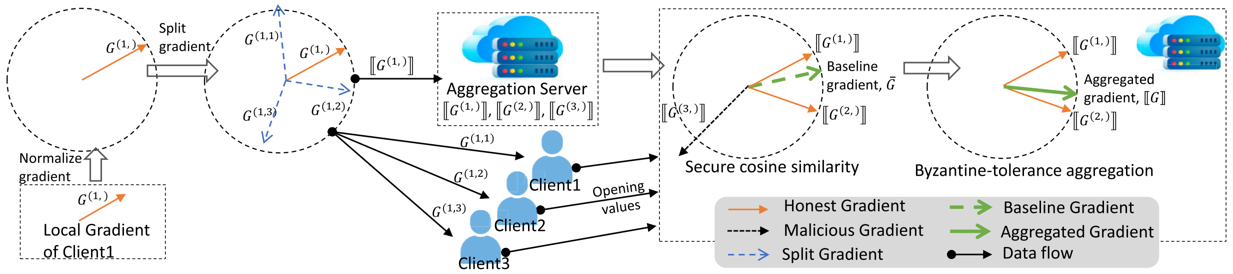

Fig. 2. Technical overview of the proposal. Each client splits its local gradient randomly into m number of gradients and shares them with other clients. Those clients use the received split gradients to produce opening values of the homomorphic encryption. At each training iteration, other clients who have $G^{(2,)}, \dots, G^{(|C|,)}$ perform similar steps with Client1 having $G^{(1,)}$.

Algorithm 1 `LocalComputing`, for a Client

```
   Input: a client's local training dataset $D$, this client's
          local root dataset $\bar{D}$, local learning rate $lr$, batch
          size $b$, the client's index $i$
1  Download global model $W_{t-1}$ from server
2  Randomly sample a batch $D' \in D$ of size $b$
3  $G_t \leftarrow \nabla L(D', W_{t-1})$; $\bar{G}_t \leftarrow \nabla L(\bar{D}, W_{t-1})$;
4  $G = G_t / \parallel G_t \parallel$
5  Randomly split $G$ into $G^{(i,1)}, \dots, G^{(i,m)}$
6  Calc encryption $[\![G]\!] \leftarrow \{[\![g_1]\!], \dots, [\![g_n]\!]\}$
7  Compute commitment $comm$ of $[\![G]\!]$, $comm'$ of $\bar{G}_t$
8  Produce $zk$-proof $\pi$ to prove the normalization of $G$
9  Send $([\![G^{(i,)}]\!], \bar{G}_t, \pi)$ to server, $(comm, comm', comm_\pi)$
   to blockchain
10 Let $R^{(i,j)}$ be randomness of $G^{(i,j)}, \forall j \in \{1, \dots, m\}$
11 Sign $G^{(i,j)}$, $R^{(i,j)}$ and send them to the $j$-th receiver for
   $j \in \{1, \dots, m\}$
```

gradients. It can prevent attacks from malicious clients who may upload large magnitudes of gradients to amplify their cosine similarity. Specifically, each client uses $G' = G / \parallel G \parallel$ to normalize its local gradient, where $\|G\|$ denotes the magnitude (i.e., Euclidean norm) of the vector G and G' denotes the unit vector.

After normalizing the gradient, the client randomly splits its local gradient $G = \{g_1, \dots, g_n\}$. For each gradient value g_k, the client generates m number of random numbers whose sum equals g_k. Concretely

$$g_k = \left\{g^{(,1)}, g^{(,2)}, \dots, g^{(,m)}\right\}, \quad \text{s.t.,} \quad g_k = \sum_{j=1}^{m} g^{(,j)} \tag{2}$$

where $g_k \in G, k \in [1, n]$. G is regarded as a vector that is randomly split into m number of vectors $\{G^{(,1)}, \dots, G^{(,m)}\}$ such that the elementwise sum of $\{G^{(,1)}, \dots, G^{(,m)}\}$ equals G. The ith client encrypts each value of the local gradient by PCS and obtains the following encryption:

$$[\![G^{(i,)}]\!] = \left\{\left[\!\left[g_1^{(i,)}\right]\!\right], \dots, \left[\!\left[g_n^{(i,)}\right]\!\right]\right\}. \tag{3}$$

The client then commits the encryption on the blockchain. Specifically, the clients build a Merkle tree [27], which has leaf nodes $[\![G^{(i,)}]\!]$. The client's commitment, which is the root of the Merkle tree, persists on the blockchain. Next, the client sends the split gradients in plaintext to the receivers, each receiving one of the various split gradients. The client also uploads all encryption to the server.

A gradient should be normalized to let the magnitudes equal one to avoid affecting the cosine similarity. Malicious clients may submit gradients with great magnitudes to amplify the clients' impacts on gradient aggregation. To let other parties verify whether the gradient has been correctly normalized, the client applies the technique of noninteractive zk-proof [28], [29], [30] to prove the following: 1) the inner product of the gradient G equal one and 2) the homomorphic encryption of value $g_k \in G$ is consistent with the one committed on the blockchain. The client proves the knowledge of G such that

$$G \circ G = 1 \wedge y^{g_k} h^{r_k} = \left[\!\left[g_k\right]\!\right] \quad \forall\, k \in [1, n] \tag{4}$$

where the symbol $\circ$ denotes the inner product; y and h are parameters of the PCS.

Algorithm 2 `CalcWeights`

```
   Input: The $j$-th receiver holds $G^{(i,j)}$, $R^{(i,j)}$ for $\forall i \in$
          $\{1, \dots, |C|\}$. The aggregation server holds
          baseline gradient $\bar{G}$ and encryption $[\![G^{(i,)}]\!]$ for
          $\forall i \in [1, |C|]$
1  The $j$-th Receiver:
2  for $i \in [1, |C|]$ do
3      $s_r^{(i,j)} \leftarrow 0$; $s_g^{(i,j)} \leftarrow 0$  //Results of inner product
4      for $k \in [1, n]$ do
5          $r \leftarrow R^{(i,j)}[k] \cdot \bar{G}[k]$;   $g \leftarrow G^{(i,j)}[k] \cdot \bar{G}[k]$
6          $s_r^{(i,j)} \leftarrow s_r^{(i,j)} + r$; $s_g^{(i,j)} \leftarrow s_g^{(i,j)} + g$
7      Send $s_g^{(i,j)}$, $s_r^{(i,j)}$ for $\forall i \in [1, |C|]$ to server and
       commit them on blockchain
8  Aggregation Server:
9  Receive $s_g^{(i,j)}$, $s_r^{(i,j)}$ for $\forall i \in [1, |C|]$ and $\forall j \in [1, m]$
10 for $i \in [1, |C|]$ do
11     Parse $[\![G^{(i,)}]\!] \rightarrow [\![g_1^{(i,)}]\!], \dots, [\![g_n^{(i,)}]\!]$
12     $[\![ip_i]\!] \leftarrow \sum_{k=1}^{n} [\![g_k^{(i,)}]\!] \cdot \bar{G}[k]$  //inner product
13     $cs_i \leftarrow \sum_{j=1}^{m} s_g^{(i,j)}$  //$i$-th client's cosine similarity
14     $r' \leftarrow \sum_{j=1}^{m} s_r^{(i,j)}$
15     Assert $[\![ip_i]\!] = y^{cs_i} h^{r'}$  //Hom. enc. of PCS
16 $w_i \leftarrow ReLU(cs_i), \forall i \in [1, |C|]$
   Output: $(w_1, \dots, w_{|C|})$  //Weights of clients
```

2) *Weight Calculation:* In this section, the server calculates cosine similarities over ciphertext for all clients at a training iteration. The clients cooperate to produce opening values to

open the ciphertext and obtain the similarities values. Those values are used as weights for secure gradient aggregation, which is described in Section III-B3. Algorithm 2 presents the process, and an explanation is given below. Each client splits its local gradient into m gradients to simplify our presentation without loss of generality. Each receiver can receive one of the split gradients, where m denotes the number of all receivers.

Given all clients' local baseline gradients, the server calculates the average to obtain the final baseline gradient. As the baseline gradient is in plaintext, the server can compute the cosine similarity for each client over homomorphic encryption (line 12). However, to open the ciphertext, the server must receive opening values from receivers. Specifically, each receiver calculates the inner product between each split gradient and the baseline gradient (similarly, calculate the inner product between the split randomness of PCS and the baseline gradient) (lines 3–6). Those inner products are then uploaded to the server. Note that the results of the inner product do not leak clients' privacy.

Malicious Case: The server will fail to open the encryption if a receiver submits a wrong result of the inner product (line 15). This scheme performs a three-step protocol presented in Section IV-C to effectively identify the malicious receiver.

The rectified linear unit (ReLU) function was applied to each cosine similarity of clients to defend against poisoning attacks from malicious clients. If a similarity value is less than zero, this client is malicious and will contribute negatively to the aggregation. Therefore, the gradient is discarded by setting the similarity value to zero. The similarity values are regarded as weights for clients to aggregate gradients. In more detail, the weights are calculated by $w_i \leftarrow \mathrm{ReLU}(s_i)$ for $i \in [1, |C|]$, and function ReLU is implemented as "$w_i = \mathrm{Max}(0, s_i)$," where s_i denotes the ith client's cosine similarity value.

Algorithm 3 `AggGrad`, to Aggregate Gradients

```
Input: The $j$-th receiver holds $G^{(i,j)}$ and $R^{(i,j)}$ for
       $\forall i \in \{1, \ldots, |C|\}$. The server holds baseline
       gradient $\bar{G}$, all clients' weights $w_i$, and
       encryption $[\![G^{(i,)}]\!]$ for $\forall i \in [1, |C|]$
1  The $j$-th Receiver:
2  Download weights $w_1, \ldots, w_{|C|}$ from server
3  for $i \in [1, |C|]$ do
4      $\hat{G}^{(i,j)} \leftarrow w_i * G^{(i,j)}$;   $\hat{R}^{(i,j)} \leftarrow w_i * R^{(i,j)}$
5  for $k \in [1, n]$ do
6      // Note, $\hat{G}^{(i,j)} = \{\hat{g}_1^{(i,j)}, \ldots, \hat{g}_n^{(i,j)}\}$
7      $g_k^{(,j)} \leftarrow \sum_{i=1}^{|C|} \hat{g}^{(i,j)}$;   $r_k^{(,j)} \leftarrow \sum_{i=1}^{|C|} \hat{r}^{(i,j)}$
8      Send $g_k^{(,j)}, r_k^{(,j)}$ for $\forall k \in [1, n]$ to server and commit
       them on blockchain
9  Aggregation Server:
10 Receive $g_k^{(,j)}, r_k^{(,j)}$ for $\forall k \in [1, n]$ and $\forall j \in [1, m]$
11 for $k \in [1, n]$ do
12     $g'_k \leftarrow \sum_{j=1}^{m} g_k^{(,j)}$;   $r'_k \leftarrow \sum_{j=1}^{m} r_k^{(,j)}$
13     //Note, $[\![G^{(i,)}]\!] = \{[\![g_1^{(i,)}]\!], \ldots, [\![g_n^{(i,)}]\!]\}$
14     Assert $\sum_{i=1}^{|C|} [\![g_k^{(i,)}]\!] \cdot w_i = y^{g'_k} h^{r'_k}$ //Hom. enc.
15     $g_k \leftarrow g'_k \cdot \parallel \bar{G} \parallel / \sum_{i=1}^{|C|} w_i$
Output: $(g_1, \ldots, g_n)$  // aggregated gradients
```

3) Model Aggregation: This section presents the process of model aggregation in Algorithm 3. Given the weights, the server performs gradient aggregation over the homomorphic encryption based on (5). In more detail, the server performs scalar multiplication of homomorphic encryption between a client's gradient and the client's weight and computes the elementwise addition of homomorphic encryption of all clients (lines 13–15), which results in an aggregated, weighted gradient but in the ciphertext

$$\begin{aligned}[\![G]\!] &= \frac{||\bar{G}||}{\sum_{i=1}^{|C|} w_i} \sum_{i=1}^{|C|} [\![G^{(i,)}]\!] \cdot w_i \\ &= \frac{||\bar{G}||}{\sum_{i=1}^{|C|} w_i} \left\{ \sum_{i=1}^{|C|} \left[\!\!\left[g_1^{(i,)} \right]\!\!\right] w_i, \ldots, \sum_{i=1}^{|C|} [\![g_n^{(i,)}]\!] w_i \right\}\end{aligned} \tag{5}$$

where w_i denotes the weight of the ith client, and G denotes the aggregated gradient. $G^{(i,)}$ denotes a normalized gradient, so the magnitude of the baseline gradient is multiplied.

To open the aggregated gradient, receivers are needed to calculate the opening values of the inner product, similar to the step in Section III-B2. Each receiver first uses a client's weight to multiply this client's split gradient (lines 3 and 4); then elementwise adds those gradients (perform similar steps for the randomness of PCS) (lines 5–7). The server uses the receivers' results to open the gradient. The server updates the global model using the aggregated gradient using the following equation:

$$W_t \leftarrow W_{t-1} - lr * G \tag{6}$$

where lr denotes the learning rate, and W_{t-1} denotes the model parameters of the previous iteration.

Malicious Case: A receiver may maliciously submit a wrong opening value. This will lead to the failure of the openness of a value in a certain dimension of the aggregated gradient. To identify this receiver, the scheme performs a three-step protocol presented in Section IV-C. Finally, our ABFL is described in Algorithm 4, which places all algorithms together into one.

Algorithm 4 `ABFL`

```
Input: global iterations $iter$, batch size $b$, clients $\{C_1,
       \ldots, C_{|C|}\}$ with training datasets $D = \{D_1, \ldots,
       D_{|C|}\}$ and root datasets $\bar{D} = \{\bar{D}_1, \ldots, \bar{D}_{|C|}\}$,
       respectively, learning rate $lr$
1  The server initializes model $W_0$
2  Clients get $W_0$ and params from server
3  for $it \in [1, iter]$ do
4      for $i \in [1, |C|]$ do
5          LocalComputing($D_i, \bar{D}_i, lr, b, i$)
6      $\{w_1, \ldots, w_{|C|}\}$←CalcWeights($\cdot$)
7      $\{g_1, \ldots, g_n\}$←AggGrad($\{w_1, \ldots, w_{|C|}\}$)
8      Server updates global model by (6)
9      Server commits $\{w_1, \ldots, w_{|C|}\}$ and $\{g_1, \ldots, g_n\}$ on
       blockchain
```

C. Other Design Considerations

Given the construction of ABFL, the other design considerations in this section are presented.

1) Audibility and Incentive: The blockchain in this system is used as a bulletin board. Thus, it is not responsible for validating the correctness of the FL process. Clients are people who benefit from the final trained global model. Thus they are motivated to ensure the correctness of the FL training. Therefore, clients are required to audit the correctness of the FL process. If more than half of the clients succeed in auditing the result based on the commitments on the blockchain, the clients can confirm that the final trained model is correctly trained. In more detail, to audit the root datasets and baseline gradients at the end of the training, a client verifies the other clients' revealed root datasets via their commitments and the preset agreement. They then recalculate those baseline gradients and compare their commitments with those that have persisted on the blockchain. To audit the aggregated gradients at the end of the training, a client performs the homomorphic encryption computation like the computation performed by the aggregation server.

Because it was assumed there more than half are honest clients in this system, all clients can confirm the correctness if half of the clients finish the audit and announce the audit result by publishing it to the blockchain. Moreover, the following incentive mechanism is applied to increase the confidence of the correctness with less than half of the clients performing the audit. Clients and servers could be required to deposit some coins to the blockchain. A client, who is the first person to find any misbehavior of another party, can produce proof and send it to the blockchain to take away that party's deposit.

2) Communication: Clients interact with one another only when a client sends its split gradients to other clients. Given an observation that the split gradients are produced randomly, the communication overhead is $O(m)$, in which the client randomly selects m number of seeds to compute the random numbers for the split gradients, then shares one of the seeds with various receivers. To satisfy the condition in (2), the sender locally keeps one split gradient such that this split gradient equals the subtraction of the original gradient and the other split gradients. The receiver can compute the same random numbers via the seed as the sender. By applying this method, the experimental results showed that the communication between clients causes negligible overheads even when the number of split gradients for a local gradient equals $|C|$ (see Section V-B).

3) Root Dataset and Baseline Gradient: To verify whether: 1) a client's local baseline gradient is generated using this client's committed local root dataset and 2) this gradient is not tampered with and honestly used to compute the final baseline gradient $\bar{G}$, the auditor performs the following two steps.

1) The auditor recomputes the local baseline gradient using this client's revealed local root dataset, and verifies the consistency via the commitment persisted on the blockchain.
2) The auditor computes the final baseline gradient $\bar{G}'$ using all newly-generated local baseline gradients for all clients. If $\bar{G}' \neq \bar{G}$, a dishonest party exists. To identify this party, the auditor compares each client's newly-generated local baseline gradient with the corresponding committed gradient, which is computed and signed by the client and stored in the server.

For the second step, two results could be obtained: One is a client's newly-generated gradient is not equal to the corresponding committed gradient. This indicates that this client tampered with the gradient. The second is that each newly-generated gradient equals the committed one but $\bar{G}' \neq \bar{G}$. This infers that the server did not honestly compute $\bar{G}$.

4) Tolerating Dropout of Clients: In the cross-silo setting, the clients are highly available. Nevertheless, a client can crash due to machine failures or communication problems. In such cases, the FL should still open the ciphertext and proceed with the training. Such user dropouts are handled by allowing each client to secretly share their m number of seeds through verifiable secret sharing (VSS) [22]. In particular, a client divides a seed into m shares, signs the shares, and secretly shares them with the receivers, such that each receiver can verify that they have received the correct share about the seed. Any k of these receivers can later recover the seed, which is used to recover the split gradient of the dropout receiver. This secret sharing introduces negligible performance overhead to the proposed system because m is a small number. In addition, a receiver holds multiple split gradients (denoted by $\{G\}$) of other clients. Thus, $\{G\}$ can be recovered if this receiver crashes. However, the server cannot aggregate the local gradient of this receiver, who is also a client. Recall that a client divides its local gradients, locally keeps one split gradient, and shares other split gradients with other clients. Therefore, this local split gradient cannot be recovered via VSS if this client crashes. To address this issue, in addition to the VSS of the m seeds, this client applies VSS to this local split gradient, which has $O(mn)$ communication complexity.

IV. Analysis

A. Correctness

To ensure that ABFL can effectively identify malicious gradients, it is important to confirm that the cosine similarity between gradients can still be correctly obtained from the encrypted data and ensure the opening values from receivers can open the homomorphic encryption to obtain the similarity and the aggregated gradient in plaintext. The following considers the case where all receivers are honest. Any misbehavior can be effectively detected and identified, which is explained in Section IV-C.

Proposition 1 (Correctness): Given the opening values from receivers who are all honest, the process `CalcWeights` in Algorithm 2 can correctly compute and open the cosine similarity between a client's local gradient and the baseline gradient.

Please refer to the proof in the Appendix.

Proposition 2 (Correctness): Given the opening values from receivers who are all honest, the process `AggGrad` in Algorithm 3 can correctly calculate and open the aggregated gradient.

Please refer to the proof in the Appendix.

B. Privacy Preservation

Recall that a client (denoted as c) randomly computes k number of split gradients, keeps one of its splits locally (denoted this split as s_{local}), and sends other $k-1$ split gradients to other clients, each of whom receives various split gradients. The c's gradient can be leaked only when k number of splits are public. This can occur only when the following conditions hold simultaneously: 1) all local split gradients of c, except s_{local}, are received from malicious clients (denoted as C_1); 2) c sends its $k-1$ splits to malicious clients (denoted as C_2); and 3) the clients in $C' \leftarrow C_1 \cup C_2$ collude. This is because, in model aggregation, each client reveals weighted averages in each gradient dimension as opening values to open the homomorphic encryption of aggregated gradient. Regarding the above first condition, those malicious clients could collude and elaborately calculate their split gradients to infer the c's local split s_{local} through the c's revealed opening values in the model aggregation process. Regarding the above second condition, those malicious clients could collude to know c's $(k-1)$ number of splits. Therefore, in the optimistic scenario where two parties do not collude, i.e., they do not leak their local data, a client's gradient keeps confidence with $k=2$.

To formally model the above issue, θ denotes the ratio of malicious clients who collude, and N denotes the number of all clients. The clients apply the above strategy to send and receive split gradients. Accordingly, the following proposition regarding the number of split gradients for a client to split its local gradient is obtained.

Proposition 3: A client's gradient keeps confidential if the client calculates k number of split gradients such that $(k-1)*2 > n*\theta$.

Proof: The client sends its $k-1$ number of splits to various clients while receiving $k-1$ number of splits from various clients. Those total $2(k-1)$ number of clients should be greater than the number of malicious clients. □

C. Identifying Misbehavior

This section analyzed malicious behaviors of clients and the server, and presented corresponding strategies to identify such a malicious party. This study assumed that the server was available during training.

In the first two cases, a lazy upload strategy was applied to improve communication and computation overheads. The encrypted split gradients were computed and uploaded to the server only when misbehavior was detected. This strategy of lazy upload makes a tradeoff in that it makes the process of identifying misbehavior from one step to the below three steps.

1) Malicious Behaviors When Opening Weights: Section III-B2 reports a malicious case where receivers could upload wrong opening values in Algorithm 2. To identify the malicious parties, each client computes homomorphic encryption of their split gradients and uploads the ciphertext to the server. In addition, a receiver calculates encryption of the received split gradient and compares the ciphertext with the ciphertext uploaded by the sender. If the two ciphertexts are not equal, the receiver uploads the split gradient in plaintext and the sender's signature to the server to prove their innocence. Given those data, the server can identify the malicious party in the following.

1) The client is malicious if the homomorphic encryption of the split gradients does not hold in criterion (2) via an additively homomorphic operation or if a client does not upload the ciphertext.
2) Given the split gradient (in plaintext) along with the sender's signature, the server first verifies the signature, then encrypts the split gradient and compares the encryption with the one uploaded by the sender. The sender is malicious if the signature verification passes but the comparison fails. The receiver is malicious if the signature verification fails.
3) If no malicious party is identified in the first two steps, the server verifies each receiver's opening values in Algorithm 2 whether $\sum_{k=1}^{n} [\![g_k^{(i,j)}]\!] \cdot \bar{G}[k]$ equals $(y^{s_g^{(i,j)}} h^{s_r^{(i,j)}})$, where $s_g^{(i,j)}$ and $s_r^{(i,j)}$ are defined in line 3 and produced by the jth receiver.

2) Malicious Behaviors When Opening Aggregated Gradient: Section III-B3 reports a malicious case where receivers upload wrong opening values in Algorithm 3, resulting in a failure to open the ciphertext in a specified dimension of an aggregated gradient. The proposed scheme performs similar steps in the previous case to identify the malicious party. First, all clients send the homomorphic encryption of the split gradient in that dimension, and all receivers reveal the corresponding plaintexts with the senders' signatures in that dimension. The server then recalculates the homomorphic encryption using those plaintexts and compares them with the on-chain commitment. Note that revealing only a 1-D gradient value does not leak a client's privacy.

3) Gradient Normalization: A client needs to use the noninteractive *zk*-proof technique to prove the normalization of the client's local gradient. A *zk*-proof has properties of completeness, soundness, and zero knowledge. Each other party can independently verify the correctness of the proof. Therefore, a *zk*-proof generated by collusive clients does not affect an honest client to verify the proof concerning the normalization result. Given the assumption that more than half of the clients are honest, incorrect proof can be detected by at least one of the honest clients if half of the clients perform the verification.

4) Malicious Server: The server may skip a client's gradient during the gradient aggregation. Alternatively, the server does not follow the FL protocol to calculate the weights or aggregate gradients, or the server may modify the intermediate result before publishing the commitment on the blockchain. Any client can reperform the computation of the server and compare the newly computed commitments with the commitments on the blockchain because all homomorphic encryptions and all intermediate results are committed on the blockchain. All ciphertexts and the results are available. In addition, if the commitments on the blockchain are traceable and immutable, the server cannot deny the misbehavior. Therefore, a client can identify these misbehaviors.

D. Communication and Computation Overhead

Clients interact with one another only when a client sends its split gradients to other clients. A client has $O(m)$ communication overheads to send and receive split gradients from other clients (see Section III-C). A client must also publish the commitments of the local gradient to the blockchain. The size of the commitments is negligible. Moreover, the client uploads homomorphic encryption of the local gradient of $O(n)$ length to the server. A client needs to upload $[s_g^{(i,j)}, s_r^{(i,j)}]$ of size $O(|C|)$ in Algorithm 2 to the server. In Algorithm 3, a client must also upload $[g_k^{(,j)}, r_k^{(,j)}]$ of size $O(n)$ to the server. The client uploads the zk-proof to the server to prove the normalization of the local gradient. The proof size depends on the concrete zero-knowledge technique. If using zkSTARKs [29], the proof size is $|\pi| = \text{O}(\log^2(N))$, where N denotes the size of an arithmetic circuit used in the proof system. If using [28] or [30], the proof size is $|\pi| = O(n)$. After the server finishes the aggregate gradients, a client downloads n number of aggregated gradients from the server. According to the above analysis, a client has a total of $O(n)$ communication overhead at each training iteration, given that $n \geq |\pi|$.

1) Computation Overhead: A client first spends $O(kn)$ time to split the local gradient and spends $T_0 = O(n)$ time to calculate homomorphic encryption for the local gradient. The client then spends $O(kn)$ time to compute $[s_g^{(i,j)}, s_r^{(i,j)}]$ in Algorithm 2 and spends the same time producing $[g_k^{(,j)}, r_k^{(,j)}]$ in Algorithm 3. Moreover, the client needs to produce the zk-proof for normalizing the gradient, which has $T_1 = O(m\cdot\text{poly-}\log(m))$ complexity when using zkSTARKs; $O(n)$ for [28] or [30]. The time complexity of this client at a training iteration is $O(nk + T_0 + T_1 + T_2 + T_3)$, where T_2 and T_3 denote the time to calculate the local gradient and the time to produce the local baseline gradient, respectively. Because the computation of splitting the gradient is over the plaintext, it is much faster than the encryption computation.

The server spends $O(n)$ time verifying the commitment of a client's gradient and the normalization of the gradient. The server spends $O(n\cdot|C|)$ time calculating the weights. The server spends $O(n\cdot|C|)$ time to aggregate gradients. Therefore, the server spends a total of $O(n\cdot|C|)$ time at a training iteration.

V. Experiments

A. Experimental Setup

1) Datasets and Setting: The MNIST dataset and Fashion-MNIST dataset were used to evaluate the performance of this scheme. Both datasets contain ten categories of images. For each dataset, 200 data (i.e., each client selects $200/|C|$ number of images) were chosen as the root dataset to calculate the baseline gradient. A private Ethereum was used to implement the blockchain.

Tensorflow 1.14 in python 3.6 was used to implement the training models. Secp256r1 elliptic curve of Crypto++ library in C++14 language was used to implement the Pedersen commitment and accelerate the computation of homomorphic encryption. SHA-256 of Crypto++ library was chosen as the hash function to produce digests and commitments. The C++ code was compiled to produce a shared library and use the python code to call the C++ functions to compute homomorphic encryption, digests, and commitments. Experiments were conducted on a cluster of machines in a local area network. Each machine was equipped with Windows 10 on an Intel Core i5-8500 CPU clocked at 3.00 GHz with 32-GB RAM.

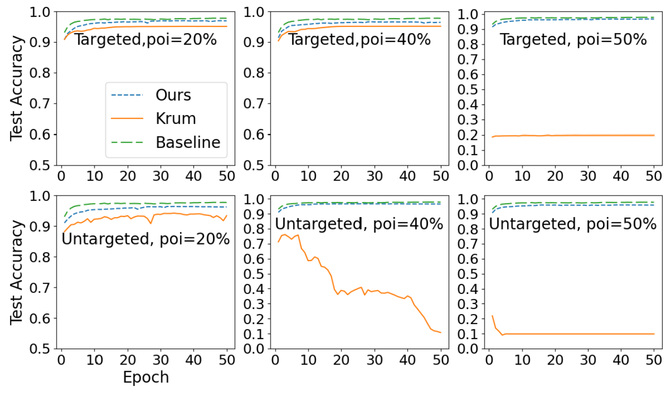

Fig. 3. Test accuracy on MNIST. "poi" denotes the ratio of malicious clients.

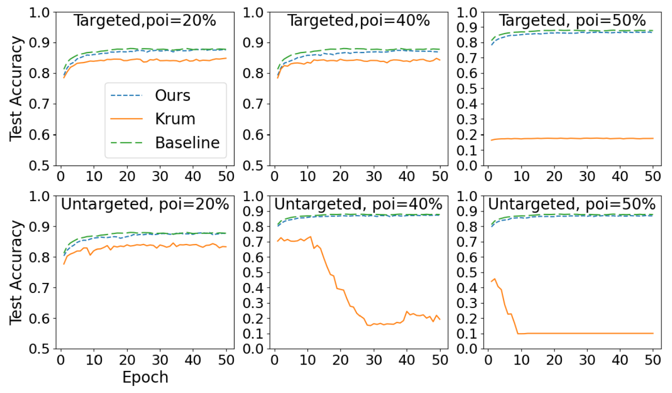

Fig. 4. Test accuracy over the Fashion-MNIST dataset.

2) Poisoning Attacks: Both targeted attacks and untargeted attacks were considered in these experiments. Label-flipping attacks were evaluated as targeted attacks. The label was flipped from l to $(l + 1 \mod m)$, where mod denotes a modulo operation, and m denotes the category number, e.g., $m = 10$ in MNIST. For untargeted attacks, it was assumed that malicious clients upload arbitrary gradients to manipulate the global model.

3) Evaluation Metrics: The test accuracy, attack success rate [31], and test error rate were used as indicators to evaluate the performance of models trained on various datasets. The attack success rate was used for targeted poisoning attacks. The test error rate was used for untargeted poisoning attacks and was calculated by the one minus average accuracy across all input samples. The FedSGD [1] without attacks was used as a baseline and Krum [11] was used as a comparison of the performance in resisting poisoning attacks.

4) FL Settings: The cross-silo setting was adopted. The number of clients was set to 10, and all clients in each iteration during training were selected. For the training model, a three-layer fully connected neural network was used to train over the MNIST and FashionMNIST datasets on Keras with a Tensorflow backend. The model has a 28×28 input shape, and the layer type of its first layer is "Flatten" with an output shape of 784. The second and third layers have a "Dense" layer type, and they have a 128 and 10 output shape, respectively. In total, the model has 101 770 trainable parameters.

The training dataset and the data were allocated equally to the clients. Thus, each client had 6000 images for MNIST

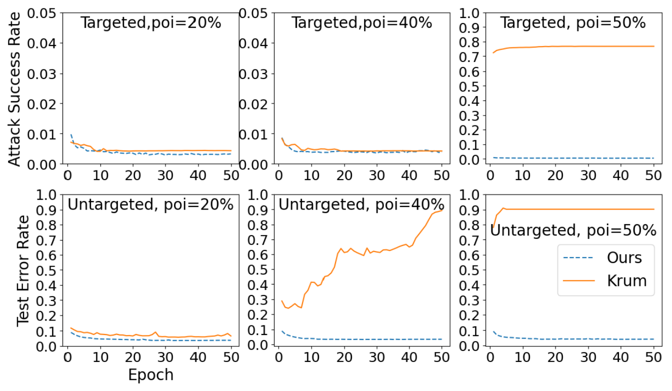

Fig. 5. Attack success rate and Attack success rate over the MNIST dataset.

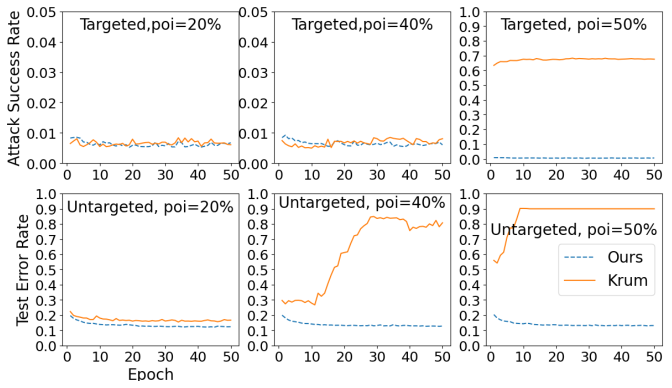

Fig. 6. Attack success rate over the Fashion-MNIST dataset.

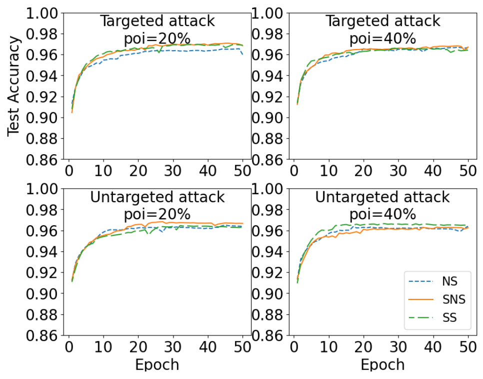

Fig. 7. Test accuracy when the proposed scheme uses the root dataset of MNIST, which is SS, or SNS, and NS.

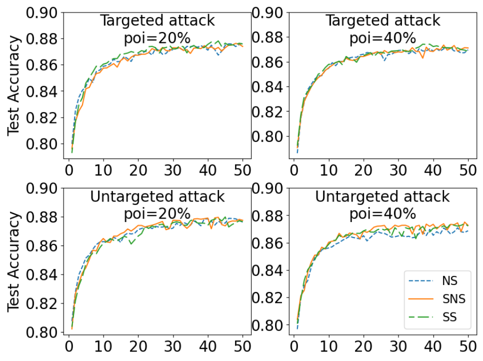

Fig. 8. Test accuracy when the proposed scheme uses various types of the root dataset of FashionMNIST.

or FashionMNIST. Moreover, the batch size was set to 100, so there were approximately 60 iterations for each epoch with ten clients. Each experimental result is the average of five repeated executions.

B. Experimental Results

The test accuracy over the MNIST dataset and Fashion-MNIST dataset was first evaluated, as shown in Figs. 3 and 4, respectively. The scheme has higher test accuracy than Krum in each case. On the MNIST and Fashion-MNIST datasets, the scheme showed similar accuracy as the baseline against both targeted and untargeted attacks. In particular, in targeted attacks, Krum lost the ability to resist targeted attacks when the ratio of malicious clients was 50%. In untargeted attacks, Krum cannot defend the attacks when the ratio is greater than and equal to 40%, whereas the scheme has stable and high performance to resist both kinds of attacks.

The test error rate and attack success rate were evaluated, as shown in Figs. 5 and 6. In both targeted and untargeted attacks, the proposed scheme has a lower test error rate and attack success rate than Krum with various ratios of malicious clients. Over both datasets, Krum has a high test error rate when the ratio of malicious clients is equal to and greater than 40% in untargeted attacks.

Recall that the root dataset is prepared by all clients. The final baseline gradient at each training iteration was calculated by averaging each client's local baseline gradient, which is obtained through each client's local root dataset. The local root dataset for a client can be selected in two ways. The first way is that each client only selects one category of samples. In the experiment, the ith client selects 20 images in the ith category. split but not shuffled (SNS) was used to denote this way. The second is that each client selects samples of all categories. In this experiment, the ith client selected 2×10 images with ten categories, each of which has two images. Split and shuffled (SS) was used to denote this way. The case where a single client prepares the whole root dataset was also applied. This way is used in prior works [14], [26]. This way is denoted by not split (NS). They were compared by evaluating their test accuracy, as shown in Figs. 7 and 8. The experimental results showed that these three ways have the same performance regarding the testing accuracy under targeted and untargeted attacks over both datasets. Moreover, Miao et al. [14] showed that the root dataset was not required to contain a large number of samples, and the root data does not require the same data distribution as the clients' local training samples. This finding reduces the difficulty for clients of this system to prepare the root dataset, even though some clients do not base the agreement on preparing their local root datasets.

1) Evaluation of Processing Time: The processing time was first evaluated to determine if the threshold Paillier algorithm can be used to replace the PCS in the proposed system, as shown in Fig. 9. Adopting the recommendation from the National Institute of Standards and Technology (NIST) [32], the security parameter was set to 3072 bit-length for the Paillier algorithm, corresponding to 256 bit-length for PCS over an elliptic curve. The "libntl" library was used to implement the Paillier algorithm in C++. Both algorithms run in a single thread. The system using the Paillier algorithm

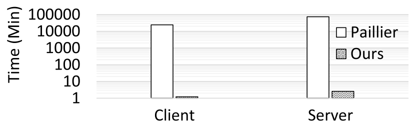

Fig. 9. Total processing time for one iteration of the training model mentioned in Section V-A, which has 101 770 trainable parameters. "Paillier" denotes the threshold Paillier algorithm [23]. The client number and the threshold of the Paillier algorithm were both set to three. For the proposed scheme, each client produced three split gradients for a local gradient.

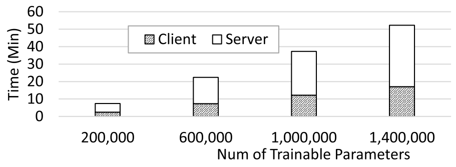

Fig. 10. Processing time for one iteration with various sizes of training models using our scheme. "One iteration" denotes the procedure from the time of producing the local gradients to the time of obtaining the corresponding aggregated gradient.

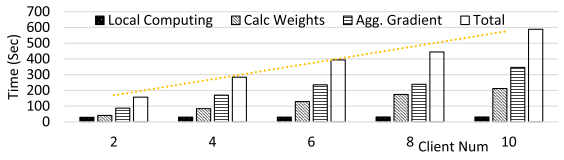

Fig. 11. Processing time for one iteration of the proposed scheme.

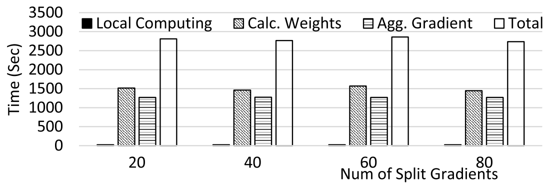

Fig. 12. Processing time for one iteration of our scheme with various numbers of split gradients for a client. We set $|C| = 100$.

was approximately 26 000× slower than the system using PCS. In addition to the large security parameter, the system that uses the Paillier algorithm has a relatively time-consuming process for threshold decryption. It needs clients to download ciphertext to partially decrypt and produce zk-proof to prove the correctness of the partial decryption. The server first verifies the proof and combines the partial decryption to obtain the plaintext.

The processing time of this scheme, in which various sizes of models are trained, was evaluated further. Fig. 10 presents the results, which indicate that the processing time of both the client and server increases linearly with larger training models. This processing time can be improved using multiple threads. The server has a longer processing time than the client. Multiple servers can be used for the load balance; those servers are not required to be trustworthy but available. Moreover, the proposed implementation code can be improved.

The processing time of the proposed system with various numbers of clients was next evaluated, as shown in Fig. 11. The result indicates that the processing time of those processes (i.e., local computing, calculate weights, aggregate gradient) increases almost linearly as the number of clients increases.

The processing time with various numbers of split gradients was evaluated to determine the effect of the number of split gradients on the processing time. The experimental result is presented in Fig. 12. The processing time of those processes remained unchanged with the greater number of split gradients. This result shows that the number of split gradients for a client has a negligible effect on the processing time. This is because the communication overhead for a client to send and receive split gradients was only $O(m)$. In addition, the computation over the plaintext, such as splitting gradients and producing opening values at clients, was much faster than the computation over ciphertext. Based on the experimental result, the proposed system can set the number of split gradients as a large value, such as $|C|$.

VI. Related Works

Blanchard et al. [11] proposed Krum, which is a Byzantine-tolerant stochastic gradient descent in FL. Krum chooses one gradient that is most likely to be benign from clients' local gradients based on the Euclidean distance to be the aggregated gradient. Yin et al. [12] calculated the aggregated gradient using a coordinatewise median or coordinatewise trimmed mean. Cao et al. [26] proposed Fltrust, which uses a small and clean root dataset to produce a baseline gradient to calculate the cosine similarity of the clients' gradient for resisting poisoning attacks. Mao et al. [33] proposed Romoa that protected FL against targeted and untargeted poisoning attacks, but the above works cannot preserve privacy.

Miao et al. [34] proposed CAFL that used compressive sensing to compress the local model for reducing communication overheads and used adaptive local differential privacy to protect privacy. But they assumed that the clients and server are semihonest and did not consider the malicious cases. Shen et al. [31] proposed Auror, which identifies malicious gradients masked by differential privacy, using the cluster algorithm KMeans. Truex et al. [35] combined differential privacy with secure multiparty computation to protect privacy. Liu et al. [16] proposed a privacy-enhanced FL scheme against poisoning adversaries. The scheme used a Paillier cryptosystem [6] to implement linearly homomorphic operations over ciphertexts to protect privacy. The Pearson correlation coefficient was used to detect malicious gradients and model the system with two servers that do not collude with each other to resist poisoning attacks while preserving privacy. With a similar architecture, Ma et al. [15] used the Paillier cryptosystem to provide two-trapdoor encryption for privacy preservation. A previous scheme [15] used the gradient, which is nearest to the aggregated gradient at the previous training round, as the baseline to calculate the cosine similarity for clients' gradients and resist poisoning attacks. The similarities were then used to calculate clients' weights to aggregate the gradients. On the other hand, the above works assume the aggregators are semihonest.

Weng et al. [19] proposed DeepChain, a distributed, auditable, and PPFL scheme that provides a value-driven incentive mechanism based on blockchain to force participants to behave correctly. On the other hand, it lacked consideration of the on-chain computation and storage overheads and could not resist poisoning attacks. The threshold Paillier algorithm [23] was used as an underlying component to secure gradients. The current experiment result showed that

TABLE I
COMPARATIVE SUMMARY BETWEEN OUR SCHEME AND EXISTING SCHEMES

Schemes	Fun_1	Fun_2	Fun_3	Fun_4	Fun_5
Krum [11]	Yes	–	No	–	–
Trim-Mean [12]	Yes	–	No	–	–
AUROR [31]	Yes	DP	No	No	Yes
CAFL [34]	No	DP	No	No	Yes
DeepChain [19]	No	Paillier	Yes	Yes	Yes
PEFL [16]	Yes	Paillier	No	No	No
PBFL [14]	Yes	CKKS	No	No	No
ADFL [36]	Yes	MaskCode	No	No	Yes
ShieldFL [15]	Yes	Paillier	No	No	No
Ours	Yes	PCS	Yes	Yes	Yes

Note, Fun_1: It denotes whether resisting poisoning attacks or not. Fun_2: Privacy-preserving mechanism. Fun_3: Whether the aggregation server is not assumed to be semi-honest, i.e., could be malicious. Fun_4: Whether achieving the auditability. Fun_5: Whether clients can collude with the servers while still preserving the other clients' privacy.

the proposed scheme based on PCS is much more efficient than the Paillier algorithm.

Miao et al. [14] proposed PBFL that uses a fully homomorphic encryption scheme to reduce computation and communication overheads and preserve privacy. It leverages the root dataset to generate the baseline gradient for defending against poisoning attacks and uses blockchain to facilitate transparent FL processes. However, PBFL uses two servers and assumes that they do not collude. If the two servers collude, they can know all clients' gradients in the clear. Furthermore, if one client colludes with the two servers, they can perform poisoning attacks without being detected. In the contrast, a client's gradient in our system keeps confidential even if the server colludes with some other parties. Also, any party in our system can audit whether the server honestly finished the aggregation.

Finally, Table I compares the present scheme with the aforementioned FL approaches. The results showed that the proposed scheme could handle cases where both the aggregator and clients are malicious. The scheme preserves privacy, resists poisoning attacks, and provides auditability against malicious participants.

VII. CONCLUSION

This article proposed an ABFL scheme against malicious but available aggregators and those clients who could perform poisoning attacks. The scheme offers auditability to enable a participant to verify the correctness and consistency of the FL process. Any misbehavior can be detected, and the corresponding malicious party can be effectively identified. Moreover, the scheme preserves client privacy and resists poisoning attacks. The technique of divide and conquer helps secure client privacy. The clients cooperate to decrypt ciphertext so that their privacy is secure even if some other clients leak their local data.

APPENDIX
TECHNICAL BACKGROUND, PROOF, AND MORE EXPERIMENTAL RESULT

A. Federated Learning

In FL, a central aggregation server will coordinate $|C|$ number of clients $\{C_1, \ldots, C_{|C|}\}$ to join and train the same ML model, such as image classification. Each client C_i holds a private local dataset D_i. All clients share the same initial parameters and hyperparameters of the learning model. Generally, a mini-batch stochastic gradient descent is used by clients to optimize the model via a loss function $L(D', W_t)$, where W_t denotes the model parameters at the tth round. The client C_i calculates the local gradient G_i of W_t by estimating $G_i \leftarrow \nabla L(D'_i, W_t), D'_i \in D_i$. Given all the clients' local gradients $\{G_1, \ldots, G_{|C|}\}$ at this round, the server performs gradient aggregation by following a predefined strategy, e.g., averaging. With the aggregated gradient G, the server updates the global model by $W_{t+1} \leftarrow W_t - lr * G$, where lr denotes a learning rate. All clients then download W_{t+1} from the server to begin a new round of training. This synchronized procedure is repeated until the global model converges or reaches the maximum number of training rounds.

B. Pedersen Commitment Scheme

Homomorphic encryption allows computation directly on encrypted data without requiring access to a secret key. The PCS [22], which has properties of perfectly hiding and computationally binding, was used to perform the additively homomorphic encryption. Specifically, let $\mathbb{G}$ be a cyclic group with $s = |\mathbb{G}|$ elements, and let h and y be two random generators of $\mathbb{G}$. The public key of the Pedersen commitment is $pk = (\mathbb{G}, h, y)$ (no secret key exists). Encryption of the Pedersen commitment is calculated by $c \leftarrow y^m h^r$ on the input message $m \in \{0, \ldots, s-1\}$ and randomness r. As for additively homomorphic property, an equation $c_1 \cdot c_2 = (y^{m_1} h^{r_1}) \cdot (y^{m_2} h^{r_2}) = y^{m_1+m_2} h^{r_1+r_2}$ if c_1 and c_2 are two encryption to message m_1 and m_2 with randomness r_1 and r_2, respectively. This additively homomorphic equation can be denoted abstractly by $[\![m_1]\!] + [\![m_2]\!] = [\![m_1 + m_2]\!]$. As for multiplication with a scalar x, $c_1^x = (y^{m_1} h^{r_1})^x = y^{m_1 \cdot x} h^{r_1 \cdot x}$, which is denoted by $[\![m_1]\!] \cdot x = [\![m_1 \cdot x]\!]$. Unless otherwise stated, the description of the randomness is skipped for simplicity.

C. *Proof for* Proposition 1

Proposition 1 (Correctness): Given the opening values from the receivers who are all honest, the process `CalcWeights` in Algorithm 2 can correctly compute and open the cosine similarity between a client's local gradient and the baseline gradient.

Proof: Based on (7), because all the encryptions are available in the server, the server computes the inner product for cosine similarity between a client's gradient $[][G^{(i,)}]\!]$ and the baseline gradient. Through (8) and (9), the receivers, who have the ith client's split gradients in plaintext, can cooperate to calculate the cosine similarity without leaking the split gradients to each other. Specifically, each receiver computes the inner product between the client's split gradient and the baseline gradient shown in (8). Given the inner products from all receivers, the cosine similarity in plaintext can be obtained using (9). This similarity in plaintext is exactly the opening of that similarity over homomorphic encryption in the following equation:

$$[\![G^{(i,)}]\!] \circ \bar{G} = \sum_{k=1}^{n} \left([\![g_k^{(i,)}]\!] \right) \cdot \bar{g}_k \tag{7}$$

$$\left\{ \begin{array}{c} \left(G^{(i,1)}\right) \circ \bar{G} \\ \cdots \\ \left(G^{(i,m)}\right) \circ \bar{G} \end{array} \right\} = \left\{ \begin{array}{c} \left\{ g_1^{(i,1)} \bar{g}_1, \ldots, g_n^{(i,1)} \bar{g}_n \right\} \\ \cdots \\ \left\{ g_1^{(i,m)} \bar{g}_1, \cdots, g_n^{(i,m)} \bar{g}_n \right\} \end{array} \right\}$$

$$= \left\{ \underbrace{\sum_{k=1}^{n} g_k^{(i,1)} \bar{g}_k}_{\text{receiver}_1}, \ldots, \underbrace{\sum_{k=1}^{n} g_k^{(i,m)} \bar{g}_k}_{\text{receiver}_m} \right\} \tag{8}$$

$$\Rightarrow \sum_{k=1}^{n} \left(g_k^{(i,1)} + \ldots + g_k^{(i,m)} \right) \bar{g}_k$$

$$= \sum_{k=1}^{n} g_k^{(i,)} \bar{g}_k = G^{(i,)} \circ \bar{G}. \tag{9}$$

□

D. *Proof for* Proposition 2

Proposition 2 (Correctness): Given the opening values from receivers who are all honest, the process `AggGrad` in Algorithm 3 can correctly compute and open the aggregated gradient.

Proof: Without loss of generality, the focus is on the computation of the first dimension of gradients. Given all encryptions, the server obtains the values of all clients' gradients in the first dimension, then calculates the inner product with the clients' weights to obtain the aggregated gradient value in the first dimension shown in (10).

To open this aggregated gradient value, each receiver computes the inner product between the split gradient and the client weights by (12). Given the results of the inner product from all receivers, the weighted aggregation of this gradient value in the first dimension is obtained using (13). This aggregation value in plaintext is exactly the opening of that aggregation value over homomorphic encryption from the following equation:

$$\left\{ [\![g_1^{(1,)}]\!], \ldots, [\![g_1^{(|C|,)}]\!] \right\} \circ \left\{ w_1, \ldots, w_{|C|} \right\}$$

$$= \sum_{i=1}^{|C|} [\![g_1^{(i,)}]\!] \cdot w_i \tag{10}$$

$$\left\{ \begin{array}{l} g_1^{(1,)} \\ \quad\cdots \\ g_1^{(|C|,)} \end{array} \right\}$$

$$\Rightarrow \left\{ \begin{array}{c} \left\{ g_1^{(1,1)}, \ldots, g_1^{(1,m)} \right\} \\ \cdots \\ \left\{ g_1^{(|C|,1)}, \cdots, g_1^{(|C|,m)} \right\} \end{array} \right\} \tag{11}$$

$$\Rightarrow \left\{ \underbrace{\sum_{i=1}^{|C|} g_1^{(i,1)} \cdot w_i}_{\text{reciever}_1}, \ldots, \underbrace{\sum_{i=1}^{|C|} g_1^{(i,m)} \cdot w_i}_{\text{receiver}_m} \right\} \tag{12}$$

$$\Rightarrow \sum_{i=1}^{|C|} \left(g_1^{(i,1)} + \cdots + g_1^{(i,m)} \right) \cdot w_i = \sum_{i=1}^{|C|} g_1^{(i,)} \cdot w_i. \tag{13}$$

□

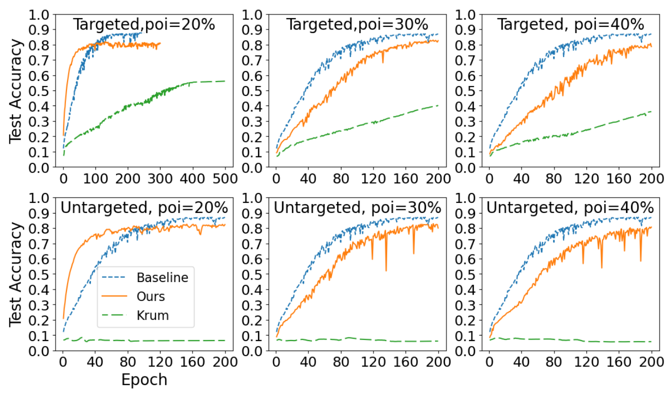

Fig. 13. Test accuracy over the chest X-ray dataset.

TABLE II
CNN Model Summary

Layer	Output Shape	Param #
Conv2D	(64, 64, 32)	832
MaxPooling2D)	(32, 32, 32)	0
Conv2D	(32, 32, 64)	18496
MaxPooling2D	(16, 16, 64)	0
Full Connection	16384	0
Full Connection	64	1048640
Dropout	64	0
Full Connection	15	975

E. *Experiments With the Chest X-Ray Dataset*

The chest X-ray dataset[1] comprises 112 120 X-ray images, 1024×1024 pixels in size, with disease labels from 30 805 unique patients. There are 15 classes, such as "No findings," "Atelectasis," and "Hernia." These classes are balanced using the function "fit_sample" of the SMOTE library, and 12 072 samples are obtained for each class. This study used 144 864 samples for training and 36 216 samples for testing. A convolutional neural network was used to train over the chest X-ray dataset. Table II lists the model summary, which contains 1 068 943 parameters.

This study evaluated the test accuracy over the chest X-ray dataset. Fig. 13 presents the experimental results. The proposed scheme showed higher test accuracy than Krum in each case. On the chest X-ray dataset, Krum fails to defend against poisoning attacks, with a poison ratio of 20%. By contrast, the proposed scheme could reach an accuracy of approximately 80% with various poison ratios. FedSGD without attacks was used as the baseline.

References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. Artif. Intell. Statist.*, 2017, pp. 1273–1282.

[2] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1333–1345, May 2018.

[3] Z. Xiong, Z. Cai, D. Takabi, and W. Li, "Privacy threat and defense for federated learning with non-i.i.d. data in AIoT," *IEEE Trans. Ind. Informat.*, vol. 18, no. 2, pp. 1310–1321, Feb. 2022.

[1]https://www.kaggle.com/datasets/nih-chest-xrays/data?resource=download

[4] K. Bonawitz et al., "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1175–1191.

[5] H. Zhu, R. Wang, Y. Jin, and K. Liang, "PIVODL: Privacy-preserving vertical federated learning over distributed labels," *IEEE Trans. Artif. Intell.*, early access, Dec. 28, 2021, doi: 10.1109/TAI.2021.3139055.

[6] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings*. Berlin, Germany: Springer, May 1999.

[7] J. Liu, J. Lou, L. Xiong, J. Liu, and X. Meng, "Projected federated averaging with heterogeneous differential privacy," *Proc. VLDB Endowment*, vol. 15, no. 4, pp. 828–840, Dec. 2021.

[8] J. Feng, Q.-Z. Cai, and Z.-H. Zhou, "Learning to confuse: Generating training time adversarial data with auto-encoder," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[9] R. Guerraoui and S. Rouault, "The hidden vulnerability of distributed learning in byzantium," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 3521–3530.

[10] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, "Data poisoning attacks against federated learning systems," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I*. Springer, 2020, pp. 480–501.

[11] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[12] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 5650–5659.

[13] J. So, B. Güler, and A. S. Avestimehr, "Byzantine-resilient secure federated learning," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2168–2181, Jul. 2020.

[14] Y. Miao, Z. Liu, H. Li, K.-K.-R. Choo, and R. H. Deng, "Privacy-preserving Byzantine-robust federated learning via blockchain systems," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 2848–2861, 2022.

[15] Z. Ma, J. Ma, Y. Miao, Y. Li, and R. H. Deng, "ShieldFL: Mitigating model poisoning attacks in privacy-preserving federated learning," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 1639–1654, 2022.

[16] X. Liu, H. Li, G. Xu, Z. Chen, X. Huang, and R. Lu, "Privacy-enhanced federated learning against poisoning adversaries," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 4574–4588, 2021.

[17] P. Ramanan and K. Nakayama, "BAFFLE: Blockchain based aggregator free federated learning," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Nov. 2020, pp. 72–81.

[18] K. Toyoda and A. N. Zhang, "Mechanism design for an incentive-aware blockchain-enabled federated learning platform," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 395–403.

[19] J. Weng, J. Weng, J. Zhang, M. Li, Y. Zhang, and W. Luo, "DeepChain: Auditable and privacy-preserving deep learning with blockchain-based incentive," *IEEE Trans. Depend. Secur. Comput.*, vol. 18, no. 5, pp. 2438–2455, Nov. 2021.

[20] Y. Zhao et al., "Privacy-preserving blockchain-based federated learning for IoT devices," *IEEE Internet Things J.*, vol. 8, no. 3, pp. 1817–1829, Feb. 2021.

[21] Y. Wan, Y. Qu, L. Gao, and Y. Xiang, "Privacy-preserving blockchain-enabled federated learning for B5G-driven edge computing," *Comput. Netw.*, vol. 204, Feb. 2022, Art. no. 108671.

[22] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology—CRYPTO'91: Proceedings*. Berlin, Germany: Springer, 2001, pp. 129–140.

[23] P.-A. Fouque, G. Poupard, and J. Stern, "Sharing decryption in the context of voting or lotteries," in *Proc. Int. Conf. Financial Cryptogr.* Springer, 2000, pp. 90–104.

[24] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," *Eur. Trans. Telecommun.*, vol. 8, no. 5, pp. 481–490, Sep. 1997.

[25] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 493–506.

[26] X. Cao, M. Fang, J. Liu, and N. Z. Gong, "FLTrust: Byzantine-robust federated learning via trust bootstrapping," in *Proc. 28th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[27] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16–20, 1987, Proceedings*. Santa Barbara, CA, USA: Springer, 1987, pp. 369–378.

[28] J. Groth, "Linear algebra with sub-linear zero-knowledge arguments," in *Advances in Cryptology—CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2009. Proceedings*. Berlin, Germany: Springer, 2009, pp. 192–208.

[29] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," Cryptol. ePrint Arch., Paper 2018/046, 2018. [Online]. Available: https://eprint.iacr.org/2018/046

[30] I. Damgård and E. Fujisaki, "A statistically-hiding integer commitment scheme based on groups with hidden order," in *Advances in Cryptology—ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security Queenstown, New Zealand, December 1–5, 2002 Proceedings*. Berlin, Germany: Springer, 2002, pp. 125–142.

[31] S. Shen, S. Tople, and P. Saxena, "AUROR: Defending against poisoning attacks in collaborative deep learning systems," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 508–519.

[32] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, *Recommendation for Key Management: Part 1—General*, Standard SP 800-57 Part 1 Rev. 5, 2020.

[33] Y. Mao, X. Yuan, X. Zhao, and S. Zhong, "Romoa: Robust model aggregation for the resistance of federated learning to model poisoning attacks," in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I*. Springer, 2021, pp. 476–496.

[34] Y. Miao, R. Xie, X. Li, X. Liu, Z. Ma, and R. H. Deng, "Compressed federated learning based on adaptive local differential privacy," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Dec. 2022, pp. 159–170.

[35] S. Truex et al., "A hybrid approach to privacy-preserving federated learning," in *Proc. 12th ACM Workshop Artif. Intell. Secur.*, Nov. 2019, pp. 1–11.

[36] J. Guo et al., "ADFL: A poisoning attack defense framework for horizontal federated learning," *IEEE Trans. Ind. Informat.*, vol. 18, no. 10, pp. 6526–6536, Oct. 2022.

Yihuai Liang received the M.S. degree in computer engineering from Pusan National University, Busan, South Korea, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Information Engineering, Inha University, Incheon, South Korea.

His research interests include blockchain, crowdsourcing, security, and database.

Yan Li received the B.S. degree from the School of Computer Science and Engineering, Chongqing University of Posts and Telecommunications, Chongqing, China, and the M.S. and Ph.D. degrees from the Department of Computer Science and Information Engineering, Inha University, Incheon, South Korea, in 2017.

She is currently an Assistant Professor with the Department of Computer Science and Information, Inha University.

Byeong-Seok Shin (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1997.

He is currently a Professor with the Department of Computer and Information Engineering, Inha University, Incheon, South Korea. His research interests include medical imaging, volume visualization, and real-time rendering.