Information Sciences 576 (2021) 288–311

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

PFLM: Privacy-preserving federated learning with membership proof

Changsong Jiang, Chunxiang Xu [*], Yuan Zhang [*]

School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, 611731, China
Yangtze Delta Region Institute (Huzhou), University of Electronic Science and Technology of China, Huzhou, 313001, P. R. China

A R T I C L E I N F O

Article history:
Received 8 March 2021
Received in revised form 23 April 2021
Accepted 30 May 2021
Available online 4 June 2021

Keywords:
Privacy-preserving
Federated learning
Machine learning
Membership proof

A B S T R A C T

Privacy-preserving federated learning is distributed machine learning where multiple collaborators train a model through protected gradients. To achieve robustness to users dropping out, existing practical privacy-preserving federated learning schemes are based on (t, N)-threshold secret sharing. Such schemes rely on a strong assumption to guarantee security: the threshold t must be greater than half of the number of users. The assumption is so rigorous that in some scenarios the schemes may not be appropriate. Motivated by the issue, we first introduce membership proof for federated learning, which leverages cryptographic accumulators to generate membership proofs by accumulating user IDs. The proofs are issued in a public blockchain for users to verify. With membership proof, we propose a privacy-preserving federated learning scheme called PFLM. PFLM releases the assumption of threshold while maintaining the security guarantees. Additionally, we design a result verification algorithm based on a variant of ElGamal encryption to verify the correctness of aggregated results from the cloud server. The verification algorithm is integrated into PFLM as a part. Security analysis in a random oracle model shows that PFLM guarantees privacy against active adversaries. The implementation of PFLM and experiments demonstrate the performance of PFLM in terms of computation and communication.

1. Introduction

Federated learning is a distributed machine learning technique to train a high-accuracy model by a set of collaborators, where each one of the collaborators has her/his own data set for training but does not disclose data to others [10,33]. Federated learning utilizes a paradigm of centralized model training on “decentralized” data, which is in accord with the data management paradigm of cloud computing [23,44]. As such, federated learning is mainly deployed in cloud to train a target model by using a rich set of data from different users, in which each user computes a gradient on her/his data set and sends the gradient to the cloud server; the latter then collects gradients from all users and provides each user with an aggregated gradient. With the aggregated gradient, each user can calculate a target model. Such a cloud-based federated learning actually has become a fundamental mode in reality and has received widespread attention from academia and industry [25,26].

Despite the protection of users’ data, federated learning still suffers from critical threats towards users’ privacy. Research works [27,28,31] have demonstrated that an adversary, e.g., a curious cloud server, is able to extract privacy information

[*] Corresponding authors at: School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, 611731, China

E-mail addresses: chxxu@uestc.edu.cn (C. Xu), zy_loye@126.com (Y. Zhang).

https://doi.org/10.1016/j.ins.2021.05.077
0020-0255/© 2021 Elsevier Inc. All rights reserved.

about a user from her/his gradient. For example, an adversary is able to deduce from gradients whether a data sample belongs to a training set [28,31]. Worse still, in some cases, the adversary can extract the data from gradients [27]. Consequently, it is inadvisable to simply use gradients for model training without proper protection [21].

A feasible way to protect gradients against being disclosed while remaining the functionality is to blind gradients with double masking [6,35]. Double masking means each user holds two categories of masks including pairwise masks and a "self-mask". The cloud server calculates the aggregated result with the blinded gradients. Double masking needs all users to remain online. However, this is not always ensured. In fact, users may drop out for various reasons. (t, N)-threshold secret sharing is leveraged to address the issue of users' dropping out [6,35]. Specifically, every user's keys for generating her/his masks are shared in the set of users. When a user quits the training, the server interacts with t users or more online to retrieve the masks associated with that user to keep the training running properly.

However, a security concern arises. If the threshold t is smaller than $\lfloor \frac{n}{2} \rfloor$ [1] (n denotes the number of users), a misbehaving server may divide the set of users into two subsets, each of which surpasses the threshold. Then the server can separately deceive each subset, claiming that other users (those in the other subset) are offline. In this way, the server can retrieve every user's masks and then recover all gradients. We refer to such an attack as deceiving attacks. To resolve this problem, the schemes proposed in [6,35] rely on a quite strong assumption: the threshold t is greater than $\lfloor \frac{n}{2} \rfloor + 1$ and only $\lceil \frac{n}{2} \rceil - 1$ users or fewer are allowed to drop out. Such a rigorous constraint of the threshold might result in a catastrophic consequence in practice. If users quit the training with a high probability due to constrained devices or unreliable networks, the cloud server may fail to aggregate gradients.

An alternative approach to resist deceiving attacks is asking the server to publicly issue which users are online. Anyone can check the information to find out whether a user is *online* or not. Obviously, the server cannot deceive users and launch deceiving attacks anymore. Note simply issuing users' names or IDs may breach users' privacy. In this paper, we first introduce membership proof for federated learning. Membership proof employs cryptographic accumulators to generate the membership proofs by accumulating user IDs. The server issues the membership proofs of online users on a bulletin board such as a public blockchain. Any online user can verify the proofs to make sure her/his *online* information is involved in the proofs. The number of online users is involved in the proofs as well. As such, membership proof can effectively prevent the server from launching deceiving attacks.

With membership proof based on public blockchain [34,41] and a cryptographic accumulator [3], we propose a privacy-preserving federated learning scheme. This scheme, called PFLM, releases the assumption of the threshold in existing schemes while being secure against deceiving attacks.

In addition to deceiving attacks, a misbehaving server may return randomly chosen results to users in the training to reduce computation cost. Trusted execution environments (TEE) [32] or result verification can be deployed to deal with this concern. However, TEE may incur huge monetary costs and become a bottleneck in applications. In this paper, we design a result verification algorithm using a variant of ElGamal encryption. The verification algorithm enables users to verify the correctness of aggregated results from the server, and thereby guarantee the correctness of the training model. The verification algorithm is integrated into PFLM as a part.

We present a comprehensive security analysis for PFLM, which demonstrates that attackers cannot get any useful information of the user input even if the server colludes with multiple users in the active adversary model. The implementation of PFLM (Our code is available at https://github.com/JiangChSo/PFLM) and the experiments show the performance of PFLM in terms of computation and communication.

The remainder of this paper is organized as follows. In Section 2, we review the related work. We describe the problem statement in Section 3 and present the preliminaries in Section 4. In Section 5 and 6, we overview and propose PFLM, respectively. Then, we analyze the security of PFLM in Section 7 and evaluate its performance in Section 8. Finally, we draw the conclusions in Section 9.

2. Related work

Most federated learning schemes are based on neural networks to train their models. Typical examples include privacy setting recommendation [36,37] and pattern recognition [11,39]. Recently, emerging neural network techniques have been proposed for federated learning. Yu et al. [37] adopted neural networks to develop a tool called iPrivacy, which is able to recognize human objects for the determination of privacy sensitiveness levels. IPrivacy automates the process of privacy settings during image sharing. Considering users' social behavior (i.e., user trustworthiness), they further developed an approach built on neural networks to recommend fine-grained privacy settings for social image sharing [36]. In terms of pattern recognition, Cui et al. [11] proposed a knowledge augmented framework for joint action unit (AU) detection and facial expression recognition (FER) based on neural networks. The framework improves performance for both FER and AU detection. Zhang et al. proposed a semantics-guided neural network for skeleton-based action recognition, which enhances the feature representation capability. In practice, federated learning is confronted with threats towards users' privacy. To solve this issue, privacy-preserving federated learning emerges.

[1] $\lfloor \frac{n}{2} \rfloor$denotes the largest integer smaller than $\frac{n}{2}$.

Privacy-preserving federated learning enables the gradient to be protected with cryptographic algorithms [22]. Currently, there are mainly three strategies to protect the gradient. The first strategy is homomorphic encryption [27]. Due to the homomorphism of addition and multiplication, homomorphic encryption enables the cloud server to aggregate the encrypted gradients from users with no privacy leakage. The second one is secure multi-party computing (SMC), where participants execute multiple interactions to complete target calculations while protecting local data [6,20]. Differential privacy, the third strategy, requires users to add noises to local data before uploading the data, which guarantees the confidentiality of data [1,30].

Phong et al. [27] employed homomorphic encryption to present a privacy-preserving deep learning scheme, which achieves the same accuracy to ordinary deep learning. Nevertheless, since large-scale users or high-dimensional data will cause considerable computation costs, the scheme is not suitable for most applications. To reduce the computation costs, Bonawitz et al. [6] put forward a practical and secure federated deep learning scheme PPML by exploiting SMC to protect local gradients. Considering constraints of users' equipment and networks, they also utilized secret sharing along with key agreement to deal with users dropping out of the training. However, the interactions may incur high communication costs. Recently, Yu et al. [38] proposed a differential private approach for training neural networks by employing concentrated differential privacy. Such an approach alleviates the pressure on computation and communication costs. However, research results show that the schemes based on differential privacy are vulnerable to attacks launched by GAN networks [19].

On the other hand, to prevent adversarial servers from returning forged aggregated results, several approaches [17,32,35] have been presented. Ghodsi et al. [17] proposed a scheme called SafetyNets, which supports users to verify the correctness of results computed by the server. The scheme leverages a specialized interactive proof for verifiable execution. However, SafetyNets is only suitable for a special class of deep neural networks that can be represented as arithmetic circuits. Later, Tramer et al. [32] designed a verifiable scheme called Slalom for efficient DNNs evaluation. Slalom works with trusted execution environments such as Intel SGX, ARM TrustZone, and Sanctum. Nevertheless, additional hardware support causes more costs. Recently, Xu et al. [35] used SMC to present a scheme called VerifyNet for securely aggregating gradients in the honest but curious setting. VerifyNet exploits homomorphic hash function and pseudorandom technology to achieve verifiability for users.

Existing schemes, such as PPML and VerifyNet, introduce extra constraints on the threshold to resist deceiving attacks. In this paper, we propose a privacy-preserving federated learning scheme with membership proof called PFLM, which is secure against deceiving attacks and removes the constraints. PFLM provides verifiability of aggregated results returned by the server with accepted overhead. Our scheme guarantees the privacy protection through blinding gradients and remains the robustness to users dropping out.

3. Problem statement

In this section, we describe the main concepts of federated learning, the system model, the threat model, and design goals.

3.1. Federated learning

Federated learning enables users to generate gradients, which serve as interactive information with a server. During each iteration, the server collects the gradients and returns global parameters after aggregation operations. Users wait for the server to return the aggregated parameters to update local gradients. Repeating the iteration steps, once the current training satisfies convergence conditions, users can obtain an optimal model and the relevant parameters.

Neural networks, as the underlying architecture of federated learning, achieve massive tasks such as classification, prediction, and regression. In general, neural networks can be represented as $f(\boldsymbol{x}, \boldsymbol{\theta}) = \hat{y}$, where inputs $\boldsymbol{x}$ are mapped to outputs $\hat{y}$ via function f with parameters $\boldsymbol{\theta}$. When training a model, a loss is defined as $\mathcal{L}_f(\mathcal{D}, \boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|}\sum_{(\boldsymbol{x_i}, \boldsymbol{y_i})\in\mathcal{D}}\mathcal{L}_f(\boldsymbol{x_i}, \boldsymbol{y_i}, \boldsymbol{\theta})$ on a training set $\mathcal{D} = \{(\boldsymbol{x_i}, \boldsymbol{y_i}), i = 1, \cdots, T\}$, where $\mathcal{L}_f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = l(\boldsymbol{y}, f(\boldsymbol{x}, \boldsymbol{\theta}))$ for a loss function l, e.g., $l(\boldsymbol{y}, f(\boldsymbol{x}, \boldsymbol{\theta})) = l(\boldsymbol{y}, \hat{y}) = ||\boldsymbol{y}, \hat{y}||_2$ and $||\cdot||_2$ is the l_2 norm of a vector. Training a neural network aims to find the parameters $\boldsymbol{\theta}$ that minimize $\mathcal{L}_f(\mathcal{D}, \boldsymbol{\theta})$. In PFLM, stochastic gradient descent [8] is adopted to achieve the task. Parameters are updated as follows.

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta\Delta\mathcal{L}(\mathcal{D}^k, \boldsymbol{\theta}^k),$$

where $\boldsymbol{\theta}^k$ denotes the parameters after iteration k, η is the learning rate, and $\mathcal{D}^k$ is a subset of $\mathcal{D}$ selected randomly during iteration k.

In the federated learning setting, each user $U_n \in \mathcal{U}$ holding a private set $\mathcal{D}_n$ trains a certain neural network model with her/his local data. During iteration k, a random subset of users $\mathcal{U}^k \subseteq \mathcal{U}$ is selected by the server. Then, each user $U_n \in \mathcal{U}^k$ chooses a random subset $\mathcal{D}_n^k \subseteq \mathcal{D}_n$ to execute stochastic gradient descent. Accordingly, the server calculates the desired parameters as below.

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta\frac{\sum_{U_n\in\mathcal{U}^k}\rho_n^k}{\sum_{U_n\in\mathcal{U}^k}|\mathcal{D}_n^k|},$$

where $\rho_n^k = |\mathcal{D}_n^k|\Delta\mathcal{L}_f(\mathcal{D}_n^k, \theta^k)$ is calculated by U_n and then shared to the cloud server. Ultimately, the server returns the renewal of parameters to all users.

3.2. System model

The system model of PFLM is shown in Fig. 1. There are three entities: users, each of which has its own local dataset and aims to train a high-accuracy model; cloud server, which has abundant storage space and powerful computing capability; trusted authority (TA), who initializes the system parameters and distributes keys to users and the cloud server.

Each user pre-processes its local dataset to generate gradients. Then, the gradients will be masked and sent to the cloud server along with the proof of result verification for training the final model. To resist deceiving attacks, the cloud server needs to generate membership proofs and record them into a public blockchain (e.g., Ethereum blockchain) for users to verify. The cloud server is also responsible for forwarding the proofs of result verification, aggregating the masked gradients uploaded by users, and returning the aggregated result. According to the received messages, users can verify the correctness of the aggregated result.

3.3. Threat model and design goal

3.3.1. Threat model

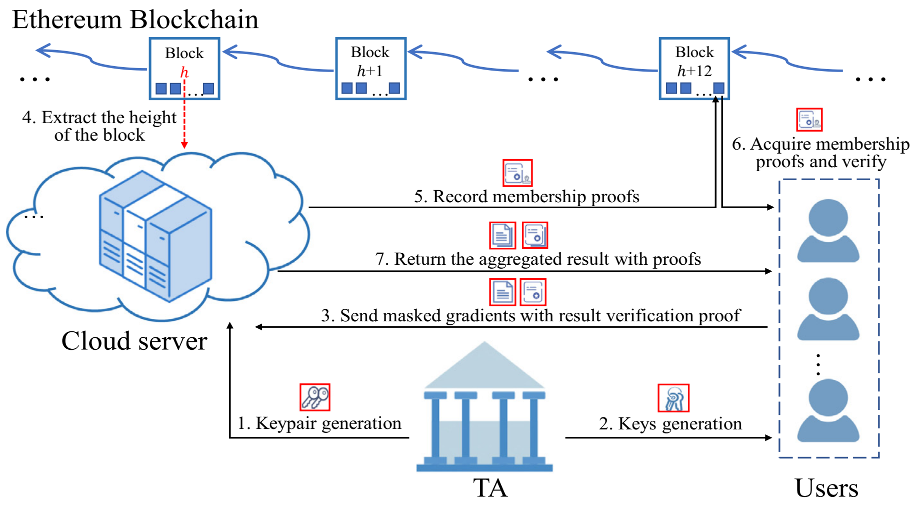
The threat models of privacy-preserving federated learning scheme can be categorized into two types: honest-but-curious security model and active adversary model [6]. In the honest-but-curious security model, both the cloud server and users are following the protocol honestly, but they may collude with others and try to infer users' data privacy. By contrast, the parties in the active adversary model not just have the above capabilities, but may deviate from the protocol (e.g., sending incorrect or randomly chosen messages, aborting, and omitting messages). Therefore, as long as the scheme is secure in the active adversary mode, it can ensure security in all models.

In this paper, we consider the active adversary model. We assume all participants including the server are active adversaries, who can deviate from the protocol and infer the privacy of user data. Furthermore, we allow that the cloud server colludes with users to get the most offensive capabilities, and also allow the server to forge the aggregated results. In PFLM, we stress that the malicious server can launch an attack called deceiving attacks to obtain users' gradients. Deceiving attacks are described as follows.

Deceiving attacks. Let $\mathcal{U}$ be a set of users. If the threshold of secret sharing is not more than $\frac{|\mathcal{U}|}{2}$, an adversarial server may divide $\mathcal{U}$ into two subsets denoted by $\mathcal{U}_1$ and $\mathcal{U}_2$, each of which surpasses the threshold. The server can deceive $\mathcal{U}_1$ that users in $\mathcal{U}_2$ are offline, and also inform $\mathcal{U}_2$ that users in $\mathcal{U}_1$ have dropped out. By doing so, the server can reconstruct every user's masks which are leveraged to blind gradients and then recover all gradients.

3.3.2. Design goal

In this paper, we target to construct a secure and verifiable federated learning scheme, which achieves the following security and performance guarantees.



Ethereum Blockchain
Block h
Block h+1
Block h+12
4. Extract the height of the block
5. Record membership proofs
6. Acquire membership proofs and verify
7. Return the aggregated result with proofs
3. Send masked gradients with result verification proof
Cloud server
1. Keypair generation
2. Keys generation
TA
Users

Fig. 1. System model of PFLM.

1. *Data privacy*. The gradients of users need to be secure against multiple participants colluding, even if the cloud server colludes with users.
2. *Verifiability*. To prevent a malicious server from returning incorrect aggregated results, each user needs to verify the correctness proofs of aggregated results.
3. *Robustness to failures*. In reality, users always drop out of the scheme due to unpredictable reasons. For practicality, our scheme needs to tolerate users dropping out in the training.
4. *Resistance against deceiving attacks*. An adversarial server may retrieve users' gradients by performing deceiving attacks. The proposed scheme should resist deceiving attacks to ensure the security.

4. Preliminaries

4.1. Bilinear maps

Let $\mathbb{G}_1$ be an additive cyclic group, and $\mathbb{G}_2$ is a multiplicative cyclic group. $\mathbb{G}_1$ and $\mathbb{G}_2$ have the same prime order q. A bilinear map [42] is that $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ with the following properties:

1. Bilinear: $\hat{e}(aQ, bR) = \hat{e}(Q, R)^{ab}$ for any $Q, R \in \mathbb{G}_1$ and $a, b \in Z_q^*$.
2. Non-degeneracy: $\hat{e}(Q, R) \neq 1$ for any $Q, R \in \mathbb{G}_1$ and $Q \neq R$.
3. Computability: there exists an efficiently computable algorithm to compute $\hat{e}(Q, R)$ for any $Q, R \in \mathbb{G}_1$.

4.2. Identity-based aggregate signature

The identity-based aggregate signature protocol [16] consists of five algorithms. **IBAS.setup** $\rightarrow \{\mathbb{G}_1, \mathbb{G}_2, \hat{e}, P, Q, H_1, H_2, H_3, s\}$ produces system parameters: groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of prime order q, a bilinear map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$, a generator $P \in \mathbb{G}_1, Q = sP$ where s is the master key, and three cryptographic hash functions $H_1, H_2 : \{0,1\}^* \rightarrow \mathbb{G}_1, H_3 : \{0,1\}^* \rightarrow Z_q$. **IBAS.pkg**$(s, U_n) \rightarrow \{sP_{n,j}, j \in \{0,1\}\}$ takes s and the user's ID U_n as input, outputs the public key $sP_{n,j}, j \in \{0,1\}$ for each user U_n, where $P_{n,j} = H_1(U_n, j) \in \mathbb{G}_1$. **IBAS.sign**$(w, M_n, U_n) \rightarrow \sigma_n$ allows that any part U_n signs M_n to obtain the individual signature $\sigma_n = \{w, S'_n, T'_n\}$, where w is the hash value of the current session ID in PFLM. Each one first calculates $P_w = H_2(w) \in \mathbb{G}_1$ and $c_n = H_3(M_n, U_n, w) \in Z_q$, then computes $S'_n = r_n P_w + sP_{n,0} + c_n sP_{n,1}$ and $T'_n = r_n P$ with a randomly chosen $r_n \in Z_q$. **IBAS.agg**$(\{U_m, \sigma_m\}_{U_m \in \mathcal{U}}) \rightarrow \{w, S_w, T_w\}$ allows each user to aggregate a collection of individual signatures to obtain the aggregate signature $\{w, S_w, T_w\}$, where $\mathcal{U}$ denotes the set of users' ID, $|\mathcal{U}| = N, S_w = \sum_{m=1}^{N} S'_m$ and $T_w = \sum_{m=1}^{N} T'_m$. **IBAS.ver**$(\mathcal{U}, w, S_w, T_w, \{M_m\}_{U_m \in \mathcal{U}}) \rightarrow \{0,1\}$ outputs 1 if $\hat{e}(S_w, P) = \hat{e}(T_w, P_w)\hat{e}(Q, \sum_{m=1}^{N} P_{m,0} + \sum_{m=1}^{N} c_m P_{m,1})$, otherwise 0, where $P_{m,j} = H_1(U_m, j), P_w = H_2(w)$, and $c_m = H_3(M_m, U_m, w)$ as above.

4.3. Secret sharing

Shamir's (t, N)-threshold secret sharing protocol [29] divides a secret s into N separate parts. s can be easily reconstructed from any t pieces and cannot be revealed even complete knowledge of $t - 1$ pieces. Specifically, (t, N)-threshold secret sharing protocol involves the following steps.

1. **S.share**$(s, t, \mathcal{U}) \rightarrow (U_n, s_n)_{U_n \in \mathcal{U}}$: The sharing algorithm takes as input a secret s, a set $\mathcal{U}$ representing the set of users' ID (presumed to be distinctive) specified in a finite field $\mathcal{F}$, and a threshold $t \leqslant |\mathcal{U}|$, and outputs the share s_n of s for each user U_n.
2. **S.recon**$(\{(U_n, s_n)\}_{U_n \in \mathcal{M}}, t) \rightarrow s$: The reconstruction algorithm takes as input the threshold t and the shares corresponding to a subset $\mathcal{M} \subset \mathcal{U}$ such that $|\mathcal{M}| \geqslant t$, and outputs the secret s.

4.4. Key agreement

Diffie-Hellman key agreement protocol [12] consists of a tuple of algorithms $\{$**KA.param**, **KA.gen**, **KA.agree**$\}$. The first algorithm **KA.param**$(k) \rightarrow pp$ produces some public parameters $pp = \{G, g, q\}$, where G is a group with the prime order q and g is the generator of G. The second algorithm **KA.gen**$(pp) \rightarrow (SK_n, g^{SK_n})$ allows any party U_n to generate a private–public key pair. **KA.agree**$(SK_n, g^{SK_m}) \rightarrow s_{n,m}$ allows any user U_n to combine their private key SK_n with the public key g^{SK_m} for user U_m (generated using the same public parameters), to obtain a private shared key $s_{n,m}$. In real-world applications, $s_{n,m}$ is often set to $H((g^{SK_m})^{SK_n})$ for convenience.

4.5. Blockchain

A blockchain can be considered as a distributed database that stores documents or digital events and shares among participating parties [41,45]. It consists of multiple data elements called blocks, which are linked to form a chain maintained by a group of participants. The participants who aim to achieve a consensus on new blocks and securely append them to the blockchain without trusting others are called miners.

Bitcoin [24] is the first application of blockchain, and another significant application is Ethereum [34] which is utilized to construct PFLM. A simplified Ethereum blockchain is described in Fig. 2, in which a block consists of two parts of data. The first one is called the block header used to compute the hash value of the current block, where *Previous block hash* serves as a pointer which points to the previous block, *Nonce* is a solution of a given Hash puzzle which constantly changes with the security requirements, *Timestamp* denotes a physical time when the block was added to the blockchain, *Tx* denotes a transaction, and *Merkle root* is the root value of the Merkle hash tree computed from all transactions in the current block.

The second one is called the transaction data containing all transactions in the current block. A graphical transaction in Ethereum is shown in Fig. 3. In general, there are two types of accounts that have a 20-byte address: externally owned accounts and contract accounts [9] in Ethereum. Fig. 3 presents the transaction from the payer's address (externally owned account) to other accounts, including the externally owned ones and contract ones. In Ethereum, a smart contract is a computer program that runs on the blockchain, and it contains the program code. Once the next block is generated, the smart contract is triggered and the associated code is executed. While the code is executing, the smart contract can read from or write to its storage file. After the next 12 blocks are generated, the current transaction is successfully published and accepted by the public [15].

4.6. Cryptographic accumulators

Cryptographic accumulators constitute one-way membership functions, which are used to verify whether a candidate belongs to a set [3]. Cardinality-proving accumulator protocol (CARDIAC) [3] leverages a Merkle tree to efficiently provide membership proofs and (non-public) proofs of maximum set cardinality. Fig. 4 describes a construction of the membership proofs via CARDIAC, where $l_{U_n}^{SID} = W(SID, U_n, w), H_0 = W(0)$, W is a hash function, *SID* denotes a session ID of current iteration, U_n is a user ID, and w is the hash value of *SID*. In Fig. 4, five users participating in the training are represented by non-zero leaf nodes, which are colored with orange. The other green leaf nodes represent zero leaf nodes. The sibling path of $l_{U_1}^{SID}$ is described by the red dotted line, and the sibling path of the rightmost non-zero leaf node $l_{U_5}^{SID}$ is depicted by the orange dotted line.

5. Overview of PFLM

A set of users, a cloud server, and a trusted authority TA are involved in PFLM. In Fig. 5, we summarize PFLM, which consists of five rounds and is described in the following.

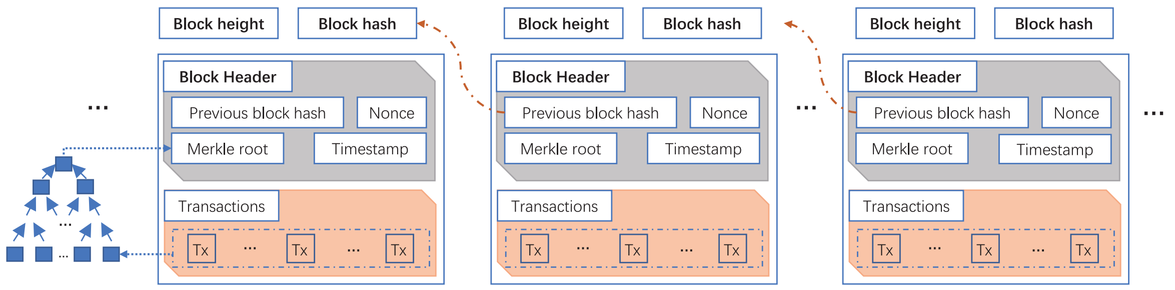
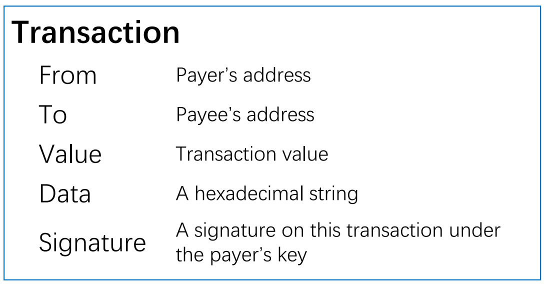


Fig. 2. A simplified Ethereum blockchain.



Transaction	
From	Payer's address
To	Payee's address
Value	Transaction value
Data	A hexadecimal string
Signature	A signature on this transaction under the payer's key

Fig. 3. A graphical transaction in Ethereum.

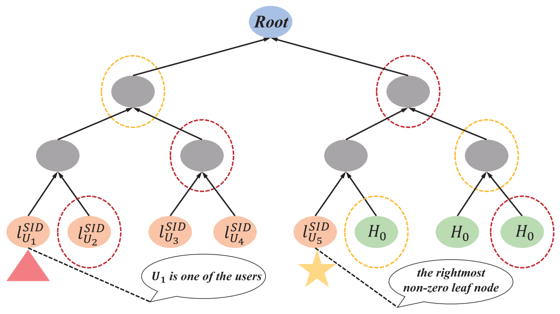


Fig. 4. Example of constructing CARDIAC.

Setup. The system is initiated, the system parameters are generated, and the parameters are distributed to all entities in the scheme. TA generates keys for users and the cloud server.

Round 0 (*Key Advertising*). TA generates the public–private key pairs used to mask users' gradients and the secret key exploited in result verification for users. The cloud server forwards users' public keys to each one.

Round 1 (*Key Sharing*). Each user shares secrets (e.g., the private key and the random seed) to others, which are served as masks. The cloud server leverages random numbers chosen by users and their IDs to construct session ID for the current iteration.

Round 2 (*Masked Input*). Each user masks local gradients with her/his secrets and generates proofs for result verification. The cloud server forwards the proofs to all users and receives their masked gradients. To resist deceiving attacks, the server records the membership proofs of online users to an Ethereum blockchain.

Round 3 (*Unmasking*). Each user verifies the correctness of the membership proofs acquired from the blockchain. If the verification passes, users learn the online users and then send relevant shares of secrets. According to the masked gradients and the shares, the server calculates the aggregated result and returns it to each user.

Round 4 (*Verification*). Each user decides to accept or reject the aggregated result by verifying the users' proofs, and returns to **round 0** to start a new iteration.

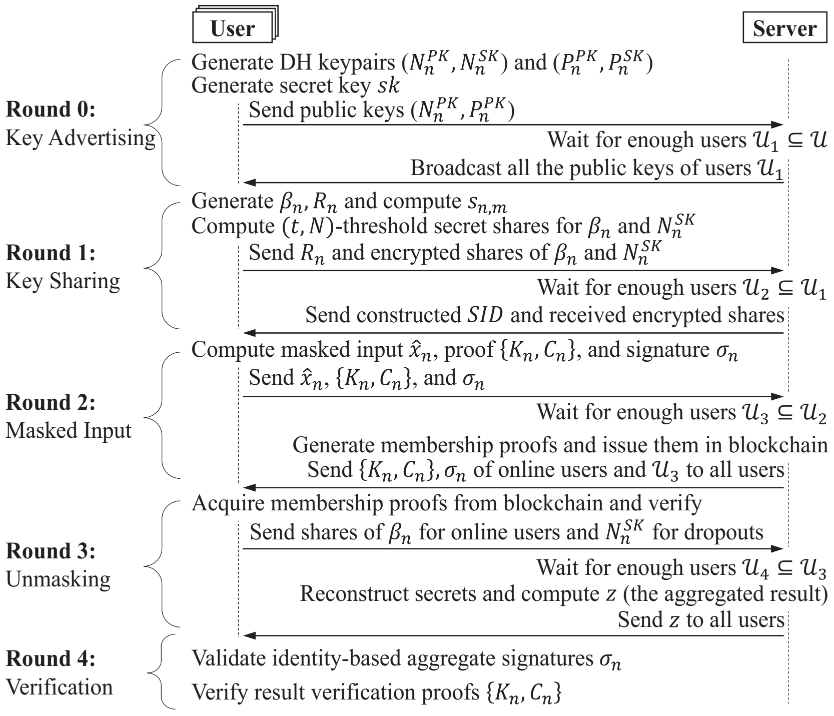

User
Server
Round 0: Key Advertising
Generate DH keypairs (N_n^{PK}, N_n^{SK}) and (P_n^{PK}, P_n^{SK})
Generate secret key sk
Send public keys (N_n^{PK}, P_n^{PK})
Wait for enough users $\mathcal{U}_1 \subseteq \mathcal{U}$
Broadcast all the public keys of users $\mathcal{U}_1$
Round 1: Key Sharing
Generate β_n, R_n and compute $s_{n,m}$
Compute (t, N)-threshold secret shares for β_n and N_n^{SK}
Send R_n and encrypted shares of β_n and N_n^{SK}
Wait for enough users $\mathcal{U}_2 \subseteq \mathcal{U}_1$
Send constructed SID and received encrypted shares
Round 2: Masked Input
Compute masked input $\hat{x}_n$, proof $\{K_n, C_n\}$, and signature σ_n
Send $\hat{x}_n$, $\{K_n, C_n\}$, and σ_n
Wait for enough users $\mathcal{U}_3 \subseteq \mathcal{U}_2$
Generate membership proofs and issue them in blockchain
Send $\{K_n, C_n\}, \sigma_n$ of online users and $\mathcal{U}_3$ to all users
Round 3: Unmasking
Acquire membership proofs from blockchain and verify
Send shares of β_n for online users and N_n^{SK} for dropouts
Wait for enough users $\mathcal{U}_4 \subseteq \mathcal{U}_3$
Reconstruct secrets and compute z (the aggregated result)
Send z to all users
Round 4: Verification
Validate identity-based aggregate signatures σ_n
Verify result verification proofs $\{K_n, C_n\}$

Fig. 5. High-level view of PFLM.

6. Proposed PFLM

We first design two building blocks: membership proof and result verification. Then, we propose PFLM based on the blocks.

6.1. Membership proof

In this section, we propose membership proof for federated learning. Specifically, the cloud server first generates membership proofs of online users via CARDIAC, then signs the root of Merkle tree and records it along with the proofs into an Ethereum blockchain. The blockchain serves as a bulletin board, which is publicly verifiable and inherently resistant to modification [41]. Given a set of user IDs $\mathcal{U}'$ ($|\mathcal{U}'| = \eta$), a session ID SID of current iteration, and a unique parameter w, $\mathbf{MP}.\mathbf{gen}(SID, \mathcal{U}', w) \rightarrow MemProof$ creates membership proofs. The details are as follows.

- Construct a Merkle tree using the element in $\mathcal{U}'$, where the non-zero leaf node of user U_n is calculated as $l_{U_n}^{SID} = W(SID, U_n, w)$.
- Compute the root of the Merkle tree $root_{SID}$ and sign it as $R_{SID} = Sig_{S_{cs}}(root_{SID})$, where S_{cs} is the cloud server's private key.
- Compute the sibling path $apm_{SID}^{U_n}$ for each $U_n \in \mathcal{U}'$.
- Get the rightmost non-zero leaf node $Rnode_{SID}$ of the Merkle tree and generate its sibling path apc_{SID}.
- Generate the membership proofs as $MemProof = \{CID, R_{SID}, \{apm_{SID}^{U_n}\}_{U_n \in \mathcal{U}'}, Rnode_{SID}, apc_{SID}\}$, where $CID = H(w, SID)$ and H is a hash function.

$\mathbf{MP}.\mathbf{rec}(\eta, Mem\ Proof) \rightarrow PubMem\ Proof$ records the membership proofs into the Ethereum blockchain as below.

- Generate an entry as $\{h, \eta, MemProof\}$, where h is the height of the newly confirmed block.
- Generate a transaction Tx_1 shown in Fig. 6, where the entry is set to the data filed. Tx_1 comes from the server's address and is sent to the address of the smart contract $contract_{cs}$.
- While collecting transactions to generate a new block whose height is $h + 12$, the miners collect the transaction Tx_1 and run the smart contract $contract_{cs}$ of the transaction Tx_1. While $contract_{cs}$ is executed, the message $\{h, \eta, MemProof\}$ in Tx_1 is written to the smart contract's storage file. Therefore, the data in the transaction Tx_1 is recorded into the public blockchain. Such data is called *PubMemProof*.

After the server records the membership proofs, each user will verify the proofs of the current iteration. Given the *PubMemProof* in the blockchain, $\mathbf{MP}.\mathbf{ver}(PubMemProof) \rightarrow \{0, 1\}$ outputs 1 if all checks pass. Otherwise, it outputs 0.

- Check the uniqueness of $\{CID, R_{SID}\}$ from *PubMemProof*.
- Acquire h and η from *PubMemProof*. Derive the height h' of the newly confirmed block under the current time. Check whether h' matches $h + 12$ and $\eta = |\mathcal{U}'|$.
- Extract $\{R_{SID}, apm_{SID}^{U_n}\}$ from *PubMemProof*. Generate the corresponding Merkle tree root $root'_{SID}$ using the sibling path $apm_{SID}^{U_n}$. Check whether $Verify(P_{cs}, root'_{SID}, R_{SID}) = true$, where P_{cs} is cloud server's public key.
- Acquire $\{R_{SID}, Rnode_{SID}, apc_{SID}\}$ from from *PubMemProof*. Generate the corresponding Merkle tree root $root''_{SID}$ using the sibling path apc_{SID} of the rightmost non-zero node $Rnode_{SID}$. Check whether $Verify(P_{cs}, root''_{SID}, R_{SID}) = true$.
- If any of the above verifications are not valid, output 0. Otherwise, output 1.

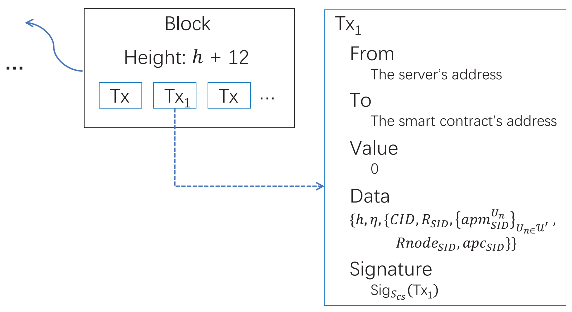

...
Block
Height: h + 12
Tx
Tx$_1$
Tx
...
Tx$_1$
From
The server's address
To
The smart contract's address
Value
0
Data
$\{h, \eta, \{CID, R_{SID}, \{apm_{SID}^{U_n}\}_{U_n \in \mathcal{U}'}, Rnode_{SID}, apc_{SID}\}\}$
Signature
$Sig_{S_{cs}}(Tx_1)$

Fig. 6. Transaction on the Ethereum blockchain.

6.2. Result verification

In PFLM, we design a result verification algorithm based on a variant of ElGamal encryption [13]. Let $\mathcal{U}$ be a set of user IDs, $|\mathcal{U}| = \eta$, and x_n the gradient of user U_n. Result verification consists of four algorithms. The first algorithm $\textbf{RV}.\textbf{param}(k) \rightarrow pp'$ takes a security parameter k as input, and outputs $pp' = \{p, g\}$, where p is a large prime number and g is a primitive root modulo p. $\textbf{RV}.\textbf{gen}(pp') \rightarrow sk$ produces a private key $sk \in Z^*_{p-1}$. $\textbf{RV}.\textbf{enc}(sk, x_n) \rightarrow \{K_n, C_n\}$ encrypts a gradient x_n by sk and a randomly chosen $k_n \in Z^*_{p-1}$, outputs the ciphertext $\{K_n, C_n\}$, where $K_n = g^{k_n} \bmod p$ and $C_n = g^{x_n} \cdot g^{k_n \cdot sk} \bmod p$. $\textbf{RV}.\textbf{ver}(z, sk, \{U_m, K_m, C_m\}_{U_m \in \mathcal{U}}) \rightarrow \{0, 1\}$ allows each user to aggregate the ciphertext $\{K_m, C_m\}_{U_m \in \mathcal{U}}$ to check whether the aggregated result z is calculated correctly. The algorithm outputs 1 if $z' = g^z \bmod p$, otherwise 0, where $z' = \prod_{m=1}^{\eta} C_m \cdot K^{-sk} \bmod p$ and $K = \prod_{m=1}^{\eta} K_m \bmod p$. In addition, the identity-based aggregate signature is employed to provide the authentication, which can convince others that user U_n owns the ciphertext $\{K_n, C_n\}$ to resist message tampering [40,43].

6.3. Construction of PFLM

In this section, we present a detailed description of PFLM in Fig. 7, which is constructed of five rounds: **Setup**, **Round 0** (*Key Advertising*), **Round 1** (*Key Sharing*), **Round 2** (*Masked Input*), **Round 3** (*Unmasking*), and **Round 4** (*Verification*).

6.4. Correctness of verification

After receiving $\{\sigma_m, U_m, K_m, C_m\}_{U_m \in \mathcal{U}_3}$ and z from the cloud server, each user first aggregates the individual signatures to get S_w and T_w, then checks whether $\hat{e}(S_w, P) = \hat{e}(T_w, P_w)\hat{e}(Q, \sum_{m \in \mathcal{U}_3} P_{m,0} + \sum_{m \in \mathcal{U}_3} c_m P_{m,1})$. Based on the Computational Diffie-Hellman (*CDH*) assumption [16], the equation holds only when each user U_n has signed the message $\{K_n, C_n, SID\}$ correctly. If it is true, any verifier is convinced that user U_n owns the relevant messages $\{K_n, C_n\}$ leveraged to construct the correctness proof of the aggregated result. Then, each user checks whether $z' = g^z$, where $z' = \prod_{m \in \mathcal{U}_3} C_m \cdot K^{-y}$ and $K = \prod_{m \in \mathcal{U}_3} K_m$. Based on the Decisional Diffie-Hellman (*DDH*) assumption [14], $z' = g^z$ holds only when $z = \sum_{n \in \mathcal{U}_3} x_n$. Finally, everyone is convinced that the server returns the correct aggregated result.

We present the correctness proof of the above equations in A. In addition, the detailed proof is omitted since it can be easily proved by utilizing *CDH* assumption [16] and *DDH* assumption [14].

7. Security analysis

In this section, we first analyze how PFLM guarantees the confidentiality of users' local gradients and resists deceiving attacks. Other security indicators are beyond the scope of this paper. Then, we summarize our scheme and show the comparison of parameterization between PPML and PFLM.

7.1. Privacy protection against active adversaries

In this section, we discuss our argument showing privacy protection against active adversaries. Active adversaries mean that the parties containing users and the server may deviate from the scheme, send incorrect or randomly selected messages to other honest parties.

Note that we only consider *input privacy* for honest users: When some users are adversarial, *correctness* and *availability* are more difficult to be guaranteed for the scheme. For instance, users can set input values x_n to be out of range, send inconsistent shares to other users in **Round 1**, or report incorrect shares to the server in **Round 3**. On the other hand, we require the server to *act honestly in the first message* (in **Round 1**), such that it honestly forwards the Diffie-Hellman public keys received from users. Therefore, users can create pairwise private and authenticated channels among themselves. Before formally presenting the complete proof, we introduce some useful definitions and notations for the active adversary setting.

We give the proofs in a random oracle $\mathcal{O}$ [4] that serves as a perfect PRG. $\mathcal{O}(x)$ outputs a *truly* random string, while $\textbf{PRG}(x)$ [5,6] outputs a pseudorandom string, where x is an input of l-bit. In this section, we assume that $\mathcal{O}$ is provided to all parties, who can make arbitrarily oracle queries to $\mathcal{O}$ and get a uniformly random result $\mathcal{O}(x)$. Besides, all honest parties will replace **PRG** calls with calls to $\mathcal{O}$ on the same input used for **PRG**.

As is standard, we considered computationally-bounded adversarial parties, which adaptively choose the set of honest parties who truly drop out of the scheme, rather than independent dropping out or predetermined aborting. The adversaries' strategies are described by probabilistic polynomial-time algorithms M. Each user U_n holds the private input x_n. We denote $x_{\mathcal{U}'} = \{x_n\}_{n \in \mathcal{U}'}$ with the inputs of any subset of users $\mathcal{U}' \subseteq \mathcal{U}$. The *view* of a party is defined as its internal state including inputs, randomness, and the messages received during the execution. Additionally, a party stops receiving messages and the view remains unchanged once it drops out.

Given any set $\mathcal{W}$ of corrupt parties, let $M_{\mathcal{W}}$ be a polynomial-time algorithm representing the "next-message" function of parties in $\mathcal{W}$. By accessing to $\mathcal{O}$, $M_{\mathcal{W}}(v, i, T, r_{\mathcal{W}})$ outputs the message for party v in round i, where T is a transcript of all mes-

Construction of PFLM

Setup :

- With the security parameter k, the system parameters $\{\mathcal{U}, t, pp, pp'\}$ are determined, where $\mathcal{U}$ is the set of users, t is the threshold, $pp \leftarrow \mathbf{KA}.\mathbf{param}(k)$, and $pp' \leftarrow \mathbf{RV}.\mathbf{param}(k)$ $(pp' = \{p, g\})$. All users have secure channels with the cloud server.

TA :

- Generate the public/private key pair (P_{cs}, S_{cs}) for the cloud server.
- Generate system parameters $\{\mathbb{G}_1, \mathbb{G}_2, \hat{e}, P, Q, H_1, H_2, H_3, s\} \leftarrow \mathbf{IBAS}.\mathbf{setup}$.
- Generate the private keys for each user U_n as $\{sP_{n,j}, j \in \{0,1\}\} \leftarrow \mathbf{IBAS}.\mathbf{pkg}(s, U_n)$.

Round 0 $(Key\ Advertising)$:

TA :

- Generate key pairs $(N_n^{PK}, N_n^{SK}) \leftarrow \mathbf{KA}.\mathbf{gen}(pp)$, $(P_n^{PK}, P_n^{SK}) \leftarrow \mathbf{KA}.\mathbf{gen}(pp)$, and generate $sk \leftarrow \mathbf{RV}.\mathbf{gen}(pp')$ for each user U_n.

$User\ U_n$:

- Send the public keys $\{N_n^{PK}, P_n^{PK}\}$ to the cloud server through a secure channel.

$Cloud\ server$:

- Receive messages from at least t users (represented as $\mathcal{U}_1 \subseteq \mathcal{U}$). Otherwise, abort and start over.
- Broadcast $\{U_m, N_m^{PK}, P_m^{PK}\}_{U_m \in \mathcal{U}_1}$ to each user $U_m \in \mathcal{U}_1$.

Round 1 $(Key\ Sharing)$:

$User\ U_n$:

- Receive the message $\{U_m, N_m^{PK}, P_m^{PK}\}_{U_m \in \mathcal{U}_1}$ from the cloud server. Check whether $|\mathcal{U}_1| \geq t$ and all the key pairs (N_m^{PK}, P_m^{PK}) are distinctive. If not, abort and start over.
- Select two random numbers β_n and R_n. Generate the shares of β_n as $\{(U_m, \beta_{n,m})\}_{U_m \in \mathcal{U}_1} \leftarrow \mathbf{S}.\mathbf{share}(\beta_n, t, \mathcal{U}_1)$, where $\beta_{n,m}$ is the share of user U_n to user U_m.
- Generate the shares of N_n^{SK} as $\{(U_m, N_{n,m}^{SK})\}_{U_m \in \mathcal{U}_1} \leftarrow \mathbf{S}.\mathbf{share}(N_n^{SK}, t, \mathcal{U}_1)$, where $N_{n,m}^{SK}$ is the share of U_n to U_m.
- Calculate $\mathcal{P}_{n,m} \leftarrow \mathbf{AE}.\mathbf{enc}(\mathbf{KA}.\mathbf{agree}(P_n^{SK}, P_m^{PK}), U_n || U_m || N_{n,m}^{SK} || \beta_{n,m})_{U_m \in \mathcal{U}_1}$, where $\mathbf{AE}.\mathbf{enc}(\cdot)$ denotes a symmetric encryption with the secret key $\mathbf{KA}.\mathbf{agree}(P_n^{SK}, P_m^{PK})$.
- Send $\{R_n, \{\mathcal{P}_{n,m}\}_{U_m \in \mathcal{U}_1}\}$ to the cloud server.

$Cloud\ server$:

- Receive messages $\{R_n, \{\mathcal{P}_{n,m}\}_{U_m \in \mathcal{U}_1}\}$ from at least t users (represented as $\mathcal{U}_2 \subseteq \mathcal{U}_1$). Otherwise, abort and start over.
- Construct session ID $SID = \{(U_1, R_1), (U_2, R_2), \cdots, (U_{|\mathcal{U}_2|}, R_{|\mathcal{U}_2|})\}$, which denotes a unique label of the current iteration.
- Broadcast $\{SID, \{\mathcal{P}_{m,n}\}_{U_m \in \mathcal{U}_2}\}$ to each user $U_n \in \mathcal{U}_2$.

Round 2 $(Masked\ Input)$:

$User\ U_n$:

- Receive $\{SID, \{\mathcal{P}_{m,n}\}_{U_m \in \mathcal{U}_2}\}$ from the cloud server. Check whether $\mathcal{U}_2 \subseteq \mathcal{U}_1$ and $|\mathcal{U}_2| \geq t$. If not, abort and start over.
- Calculate the shared key with each user $U_m \in \mathcal{U}_2$ as $s_{n,m} \leftarrow \mathbf{KA}.\mathbf{agree}(N_n^{SK}, N_m^{PK})$.
- Mask the local gradients x_n as $\hat{x}_n = x_n + \mathbf{PRG}(\beta_n) + \sum_{U_m \in \mathcal{U}_2 : U_n < U_m} \mathbf{PRG}(s_{n,m}) - \sum_{U_m \in \mathcal{U}_2 : U_n > U_m} \mathbf{PRG}(s_{m,n})$, where $\mathbf{PRG}(\cdot)$ denotes a pseudorandom generator.
- To verify the correctness of results returned from the server, some additional messages are generated as follows.
  Generate the proof by result verification as $\{K_n, C_n\} \leftarrow \mathbf{RV}.\mathbf{enc}(sk, x_n)$.
  Generate the individual signature $\sigma_n \leftarrow \mathbf{IBAS}.\mathbf{sign}(H_1(SID), \{K_n, C_n, SID\}, U_n)$, where $\sigma_n = \{w, S'_n, T'_n\}$.
- Send $\{\hat{x}_n, \sigma_n, U_n, K_n, C_n\}$ to the cloud server.

$Cloud\ server$:

- Receive messages from at least t users (represented as $\mathcal{U}_3 \subseteq \mathcal{U}_2$). Otherwise, abort and start over.
- Broadcast $\{\sigma_m, U_m, K_m, C_m\}_{U_m \in \mathcal{U}_3}$ and the list of $\mathcal{U}_3$ to each user in $\mathcal{U}_2$.
- Construct $MemProof \leftarrow \mathbf{MP}.\mathbf{gen}(SID, \mathcal{U}_3, w)$ and execute $\mathbf{MP}.\mathbf{rec}(\eta, MemProof) \rightarrow PubMemProof$, where $\eta = |\mathcal{U}_3|$.

Round 3 $(Unmasking)$:

$User\ U_n$:

- Receive $\{\sigma_m, C_m, K_m, U_m\}_{U_m \in \mathcal{U}_3}$ and the list of $\mathcal{U}_3$. Check whether $\mathcal{U}_3 \subseteq \mathcal{U}_2$ and $|\mathcal{U}_3| \geq t$. If not, abort and start over.
- Check whether $\mathbf{MP}.\mathbf{ver}(PubMemProof) \overset{?}{=} 1$. If not, abort and start over.
- Decrypt the ciphertext $\mathcal{P}_{m,n}, U_m \in \mathcal{U}_2 \backslash \{U_n\}$ as $U_m || U_n || N_{m,n}^{SK} || \beta_{m,n} \leftarrow \mathbf{AE}.\mathbf{dec}(\mathbf{KA}.\mathbf{agree}(P_m^{SK}, P_n^{PK}, \mathcal{P}_{m,n}))$.
- Send $\{(N_{m,n}^{SK}) | U_m \in \mathcal{U}_2 \backslash \mathcal{U}_3\}$ and $\{(\beta_{m,n}) | U_m \in \mathcal{U}_3\}$ to the server, where $\mathcal{U}_2 \backslash \mathcal{U}_3$ represents users who send data to the server in **Round 1**, but drop out before uploading data to the server in **Round 2**.

$Cloud\ server$:

- Collect responses from at least t users (denote as $\mathcal{U}_4 \subseteq \mathcal{U}_3$). Otherwise, abort and start over.
- Reconstruct $N_n^{SK} \leftarrow \mathbf{S}.\mathbf{recon}(\{N_{n,m}^{SK}\}_{U_m \in \mathcal{U}_4}, t)$ and $\beta_n \leftarrow \mathbf{S}.\mathbf{recon}(\{\beta_{n,m}\}_{U_m \in \mathcal{U}_4}, t)$ for each user $U_n \in \mathcal{U}_2 \backslash \mathcal{U}_3$.
- Recompute $\mathbf{PRG}(\beta_n)_{U_n \in \mathcal{U}_3}$ and $\mathbf{PRG}(s_{n,m}) \leftarrow \mathbf{PRG}(\mathbf{KA}.\mathbf{agree}(\{N_n^{SK}, N_m^{PK}\}_{U_n \in \mathcal{U}_2 \backslash \mathcal{U}_3, U_m \in \mathcal{U}_3}))$.
- Compute the aggregated gradients for all users in $\mathcal{U}_3$ as
$$\sum_{U_n \in \mathcal{U}_3} x_n = \sum_{U_n \in \mathcal{U}_3} \hat{x}_n - \mathbf{PRG}(\beta_n) - \sum_{U_n \in \mathcal{U}_3, U_m \in \mathcal{U}_2 \backslash \mathcal{U}_3 : U_n < U_m} \mathbf{PRG}(s_{n,m}) + \sum_{U_n \in \mathcal{U}_3, U_m \in \mathcal{U}_2 \backslash \mathcal{U}_3 : U_n > U_m} \mathbf{PRG}(s_{m,n}).$$
- Broadcast $z = \sum_{U_n \in \mathcal{U}_3} x_n$ to each user in $\mathcal{U}_4$.

Round 4 $(Verification)$:

$User\ U_n$:

- Compute $\{w, S_w, T_w\} \leftarrow \mathbf{IBAS}.\mathbf{agg}(\{(U_m, \sigma_m)\}_{U_m \in \mathcal{U}_3})$ and verify $\mathbf{IBAS}.\mathbf{ver}(\mathcal{U}_3, w, S_w, T_w, \{K_m, C_m, SID\}_{U_m \in \mathcal{U}}) \overset{?}{=} 1$.
- Verify $\mathbf{RV}.\mathbf{ver}(z, sk, \{U_m, K_m, C_m\}_{U_m \in \mathcal{U}_3}) \overset{?}{=} 1$.
- If any of the above equations are not valid, reject the aggregated result. Otherwise, accept the result and move to **Round 0**.

Fig. 7. Detailed description of PFLM.

sages that parties in $\mathcal{W}$ have sent or received so far, and $r_{\mathcal{W}}$ indicates the joint randomness of corrupt parties in the execution. Moreover, $M_{\mathcal{W}}(i, T, r_{\mathcal{W}})$ outputs the set of parties $\mathcal{U}_i$ that drop out due to the failure in **Round i**. Thus, such a function effectively selects the inputs for corrupt users.

A combined view of parties in $\mathcal{W}$ is denoted as a random variable $\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}(M_{\mathcal{W}}, x_{\mathcal{U}})$, where t is the threshold and k is the security parameter. In such a view, $M_{\mathcal{W}}$ chooses all messages of corrupt parties and the honest parties' independent failures, and all parties have access to $\mathcal{O}$.

In the following, we first present the theorem described the resistance against deceiving attacks. Then, two other theorems are given by considering two cases, respectively. One considers that only a subset of users is adversarial and colluding. The other is based on that the server is additionally adversarial and can collude with a subset of corrupt users.

Theorem 1. *PFLM can resist deceiving attacks from the cloud server.*

Proof. The blockchain is inherently publicly verifiable and tamper-proof. In **Round 2**, the cloud server needs to record the membership proofs of online users into the Ethereum blockchain. Once an active cloud server forges the online users to extract additional secret information, users are aware of the misbehavior by verifying the proofs in **Round 3**. Accordingly, PFLM can resist deceiving attacks from the cloud server. □

Theorem 2 (*Privacy Protection Against Joint Attacks from Active Adversaral Users, with Honest Server*). *For all PPT adversaries $M_{\mathcal{W}}$, all $t, k, \mathcal{U}, x_{\mathcal{U}\setminus\mathcal{W}}, \mathcal{W} \subseteq \mathcal{U}$, there is a PPT simulator* **SIM** *whose output is perfectly indistinguishable from the output of $\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}$:*

$$\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}(M_{\mathcal{W}}, x_{\mathcal{U}\setminus\mathcal{W}}) \equiv \textbf{SIM}_{\mathcal{W}}^{\mathcal{U},t,k}(M_{\mathcal{W}}).$$

Proof. The joint view of parties in set $\mathcal{W}$ does not depend on the inputs of the parties not in $\mathcal{W}$, since we exclude the involvement of the server. Specifically, the messages that adversarial users receive from honest users never depend on the private input x_n of those users, although they can send randomly chosen messages and decide the aborted ways of the honest users. As illustrated in **Round 2**, the response of the server does not contain the actual value of the specific $\hat{x}_n$, which means that the active adversarial users cannot identify whether the aggregated results calculated by the cloud server are based on the true inputs of honest users or dummy values. Hence, the simulator can generate a perfect simulation by using $M_{\mathcal{W}}$ for the active adversarial users on their true inputs, and all other honest users on the fake data (such as a vector of 0s). Ultimately, the simulation outputs the simulated view of the users in $\mathcal{W}$, and the output is identical to the real view of $\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}$. □

In the active adversary model, $M_{\mathcal{W}}$ dynamically decides offline users in each round. Therefore, the subset of honest users is dynamically changing, and the server learns the aggregation z from these honest users is determined dynamically. Accordingly, **SIM** is allowed to make a query to an **ideal** functionality that will learn the sum z of which from a subset L of honest parties. Formally, **SIM** can access to an oracle $\text{Idea}^{\delta}_{\{x_u\}_{u\in\mathcal{U}\setminus\mathcal{W}}}$ with appropriate δ. Given a subset L, the operation is as follows:

$$\text{Idea}^{\delta}_{\{x_u\}_{u\in\mathcal{U}\setminus\mathcal{W}}}(L) = \begin{cases} \sum_{u\in L} x_u & \textit{if } L \subseteq (\mathcal{U}\setminus\mathcal{W}) \textit{ and } |L| \geqslant \delta \\ \perp & \textit{otherwise} \end{cases}.$$

Theorem 3 presents that the joint view of corrupt parties can be simulated in the scheme if a single sum of a dynamically-chosen subset of at least δ honest users is provided.

Theorem 3 (*Privacy Protection Against Joint Attacks from Active Adversaries, including the Cloud Server*). *For all $k, t, \mathcal{U}$, and $x_{\mathcal{U}\setminus\mathcal{W}}$ such that $\mathcal{W} \subseteq \mathcal{U} \cup \{\mathcal{S}\}$, $n_{\mathcal{W}} = |\mathcal{W} \cap \mathcal{U}|$, and $n_{\mathcal{W}} < t$, there exists a PPT simulator* **SIM** *whose output is computationally indistinguishable from the output of $\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}$:*

$$\textbf{REAL}_{\mathcal{W}}^{\mathcal{U},t,k}(M_{\mathcal{W}}, x_{\mathcal{U}\setminus\mathcal{W}}) \approx_c \textbf{SIM}_{\mathcal{W}}^{\mathcal{U},t,k,\text{Idea}^{\delta}_{\{x_u\}_{u\in\mathcal{U}\setminus\mathcal{C}}}}(M_{\mathcal{W}}),$$

where $\delta = t - n_{\mathcal{W}}$.

Proof. A standard hybrid argument is utilized to prove the theorem. We will show how a simulator **SIM** executes a series of modifications to the real execution **REAL** of our scheme, which ultimately makes the view of $M_{\mathcal{W}}$ in an execution indistinguishable computationally from another one. In our hybrid argument, honest users will be dropped out by **SIM** due to their normal action in the real world, and they may be in a set $\mathcal{U}_i$ output by $M_{\mathcal{W}}$.

Hyb$_0$ The random variable which will be chosen by **SIM** is distributed exactly as the joint view of the parties $\mathcal{W}$ in the real execution of the scheme.

Hyb$_1$ In this hybrid, the simulator knowing all the inputs x_u of the honest users emulates the real execution of the scheme, and executes the scheme with $M_{\mathcal{W}}$. Concretely, **SIM** will simulate the random oracle using a dynamically generated table storing $(x, \mathcal{O}(x))$, where x is the input of querying and $\mathcal{O}(x)$ is the corresponding response to the querying. The TA and the rest of **Key Advertising** are also simulated by **SIM**. Based on the above descriptions, the view of the adversary is the same as which in the real scheme.

Hyb$_2$ In this hybrid, the messages between any two honest users m and n are encrypted and decrypted using a uniformly random number rather than the shared key $KA.agree(P_n^{SK}, P_m^{PK})$. The DDH assumption [7] guarantees that this hybrid is indistinguishable from the real scheme. Particularly, the encryption keys can be changed at a time, and an adversary cannot notice the difference unless the DDH is broken.

Hyb$_3$ In this hybrid, **SIM** will abort if $M_{\mathcal{W}}$ delivers a message to an honest user U_m on behalf of another honest user U_n successfully. Such a message is not identical with the one that **SIM** had given to M_c in **round 1**, and cannot fail the decryption algorithm when using the proper key. Note that the encryption key was selected randomly in the previous hybrid, and thereby the properties of indistinguishability under adaptive chosen-ciphertext attack (IND-CCA) security [14] guarantee the message cannot be forged.

Hyb$_4$ In this hybrid, the simulator replaces all encrypted data sent by honest users to other clients with encryption of random values, such as 0 with the appropriate length. Honest users return the "real" shares in **Round 3** as before. The properties of indistinguishability under a chosen plaintext attack (IND-CPA) [18] guarantee this hybrid is indistinguishable from the previous one in the real scheme.

Hyb$_5$ In this hybrid, the simulator additionally aborts if $M_{\mathcal{W}}$ provides the correct signature of the honest party. Due to the security of the identity-based aggregate signature based on CDH assumption [16], there is a negligible probability of forgeries, and this hybrid is indistinguishable from the previous one.

Hyb$_6$ This hybrid is defined exactly as the previous one, except that, **SIM** will abort if $M_{\mathcal{W}}$ provides the cloud server with a signature on the root of Merkle tree in **Round 2** which correctly verifies. Since the forgeries can happen with negligible probability due to the security of the signature scheme, this hybrid is identical to the real scheme.

We define a set $\mathcal{L}$ to be the only set $\mathcal{L} \subseteq \mathcal{U}$ such that the cloud server constructs the set $\mathcal{L}$ due to the responses of users in $\mathcal{U}_2$ in round **MaskedInput**, and later all users in $\mathcal{L}$ have successfully verified the membership proofs in round **Unmasking**. In case the set $\mathcal{L}$ does not exist (e.g. there are not enough users, or not enough honest users survived), we define $\mathcal{L} = \varnothing$.

Hyb$_7$ In this hybrid, **SIM** will abort if $M_{\mathcal{W}}$ inputs b_n to query the random oracle/**PRG** for the honest user U_n in the case of before the corrupt party received the responses from the honest parties in round **Unmasking** or after the responses have received but the user $U_n \notin \mathcal{L}$.

In the scheme, $M_{\mathcal{W}}$ guesses one of the b_n with a negligible probability, and **SIM** will abort if it happens. In the first case, $M_{\mathcal{W}}$ receives from **SIM** at most $n_{\mathcal{W}}$ shares of b_n sent by the corrupt user U_n in round **KeySharing**. Because $n_{\mathcal{W}} < t$, the security of (t, N)-threshold secret sharing protocol guarantees the distribution of the shares is independent from b_n. In the another case, the view of $M_{\mathcal{W}}$ is still independent from b_n, since $U_n \notin \mathcal{L}$ such that there is not honest users sending any share of b_n to the server, and therefore $M_{\mathcal{W}}$ does not receive any information from **SIM**.

Hyb$_8$ In this hybrid, the simulator aborts if $M_{\mathcal{W}}$ inputs $s_{n,m}$ to query the random oracle/**PRG** for some honest users U_n, U_m in the case of before the corrupt party received the responses from the honest parties in round **Unmasking** or after the responses have been received but the users $n, m \in \mathcal{L}$.

We can reduce the argument that this hybrid is indistinguishable from the real scheme to the security of the $2ODH$ assumption [6], which is a slight variant of the Oracle Diffie-Hellman (ODH) assumption [2]. Assuming there is another simulator **SIM**$'$ which gets a $2ODH$ challenge (G', g, q, A, B, z) and guesses randomly two honest users U_n, U_m, attempting to receive the exact $s_{n,m}$ by the adversary's query which will let **SIM**$'$ abort. **SIM**$'$ is the same as **SIM** except the public keys is set up as $N_n^{PK} = A$ and $N_m^{PK} = B$ for any pair of users, and **SIM**$'$ completes the simulation using the two oracles without the associated secret keys. Specially, the fake public keys are sent to $M_{\mathcal{W}}$ by **SIM**$'$ instead of the fresh ones sampled by **SIM** in round **Key Advertising**. In round **KeySharing**, **SIM**$'$ additionally generates shares of 0 and sends them to the corrupt users instead of the real shares of the secret keys N_n^{SK} and N_m^{SK}, which **SIM**$'$ does not know. In round **MaskedInput**, **SIM**$'$ generates $\hat{x}$ values for all honest users, replaces $s_{n,m}$ as z, and leverages the two oracles $\mathcal{O}_a$ and $\mathcal{O}_b$ to calculate all required s values for U_n and U_m and all other users. After all preparations were made, if $M_{\mathcal{W}}$ executes a random oracle query for z, **SIM**$'$ will guess $z = H(g^{ab})$ and abort; otherwise **SIM**$'$ will guess that z is a random number chosen before.

In the **2ODH** $-$ **Exp** game, in the condition of the choice of U_n, U_m being correct, the view of the adversary in the execution of simulated scheme is identical to that in **Hyb$_7$** until the adversary queries the z value by accessing a random oracle. There are two reasons to support this argument. In the previous argument, the adversary can obtain less than t shares of N_n^{SK} and N_m^{SK} for the possible values of z, so that the actual values of N_n^{SK} and N_m^{SK} cannot be revealed. Besides, $M_{\mathcal{W}}$ cannot obtain any information about $s_{n,m}$ from $\hat{x}_n$ and $\hat{x}_m$ without querying the **PRG** modeled as the random oracle.

Accordingly, if the distinction between **Hyb$_7$** and **Hyb$_8$** can be discovered by $M_{\mathcal{W}}$ with more than negligible probability, it must trigger the aborted condition (as mentioned above) with more than negligible probability and therefore $M_{\mathcal{W}}$ must query the value of form $H(g^{ab})$ by accessing to the random oracle/**PRG** with more than negligible probability. On the one hand, this means that **SIM**$'$ will assert correctly that $z = H(g^{ab})$ with a nonnegligible probability when $z = H(g^{ab})$. On the other hand, z is information theoretically hidden from the view of $M_{\mathcal{W}}$ when it is chosen uniformly random, so that $M_{\mathcal{W}}$ can only make a query for z with a negligible probability, which will let **SIM**$'$ incorrectly claim that $z = H(g^{ab})$. In summary, **SIM**$'$ would break $2ODH$ assumption with a nonnegligible probability if $M_{\mathcal{W}}$ can distinguish the difference between **Hyb$_7$** and **Hyb$_8$**, which concludes the argument.

Hyb$_9$ In this hybrid, **SIM** additionally replaces the values of $\hat{x}_n$ with the randomly selected values for all honest users and sends them to $M_{\mathcal{W}}$ in round **MaskedInput**. Besides, to ensure the consistency or correctness for the result of the scheme, **SIM** will modify the output of some random oracle queries. For the set $\mathcal{L}$ defined in **Hyb$_6$** and the user $U_n \in \mathcal{L} \setminus \mathcal{W}$, the simulator sets $\mathbf{PRG}(\beta_n)$ for the random oracle as follows:

$$\mathbf{PRG}(\beta_n) \leftarrow \hat{x}_n - x_n - \sum_{U_m \in \mathcal{K}_n} \mathbf{PRG}(s_{n,m}),$$

where $U_m \in \mathcal{K}_n$ means that an encrypted message was delivered to U_n from U_m by $M_{\mathcal{W}}$ in round **KeySharing**, which represents the fact that U_n have added the joint noise $\mathbf{PRG}(s_{n,m})$ for U_m to the masked $\hat{x}_n$. On the contrary, **SIM** replaces $\mathbf{PRG}(\beta_n)$ with random value arbitrarily for all $U_n \notin \mathcal{L} \setminus \mathcal{W}$.

Note that, $M_{\mathcal{W}}$'s view in this hybrid is indistinguishable from that in the real scheme. Because $M_{\mathcal{W}}$ cannot query the **PRG** on input β_n for the honest users $U_n \notin \mathcal{L}$, the value of $\hat{x}_n$ is uniformly and randomly distributed. And for honest users $U_n \in \mathcal{L}, \hat{x}_n$ is uniformly random because $M_{\mathcal{W}}$ cannot query **PRG** on input β_n before round **Unmasking**. And after this round, its distribution is identical to that in the real scheme as $M_{\mathcal{W}}$ masters β_n, i.e. it satisfies the following formula:

$$\hat{x}_n - \mathbf{PRG}(\beta_n) - \sum_{U_m \in \mathcal{K}_n} \mathbf{PRG}(s_{n,m}) = x_n.$$

Therefore, this hybrid is indistinguishable from the previous one.

Hyb$_{10}$ In this hybrid, for each user U_n in the set $\mathcal{L} \setminus \mathcal{W}$, the simulator sets

$$\mathbf{PRG}(\beta_n) \leftarrow \hat{x}_n - D_n - \sum_{U_m \in \mathcal{K}_n \setminus \mathcal{L} \setminus \mathcal{W}} \mathbf{PRG}(s_{n,m}),$$

instead of

$$\begin{aligned}\mathbf{PRG}(\beta_n) & \leftarrow \hat{x}_n - x_n - \sum_{U_m \in \mathcal{K}_n} \mathbf{PRG}(s_{n,m}) \\ & = \hat{x}_n - x_n - \sum_{U_m \in \mathcal{L} \setminus \mathcal{W}} \mathbf{PRG}(s_{n,m}) \\ & \quad - \sum_{U_m \in \mathcal{K}_n \setminus \mathcal{L} \setminus \mathcal{W}} \mathbf{PRG}(s_{n,m}),\end{aligned}$$

where $\{D_n\}_{U_n \in \mathcal{L} \setminus \mathcal{W}}$ is the random value selected by the simulator, and subjected to $\sum_{n \in \mathcal{L} \setminus \mathcal{W}} D_n = \sum_{U_n \in \mathcal{L} \setminus \mathcal{W}} x_n$. Therefore, the values are distributed identically as those in the real scheme.

Hyb$_{11}$ In this hybrid, the simulator additionally does not receive the honest users' inputs, but queries the functionality **Ideal** for the users in $\mathcal{L} \setminus \mathcal{W}$ and samples the required D_n using the corresponding values. In the scheme, since $|\mathcal{L}| \geqslant t$ and $|\mathcal{L} \setminus \mathcal{W}| \geqslant t - n_{\mathcal{W}}$ are satisfied, **Ideal** will not return $\perp$.

Therefore, the adversary's view has not been modified by this change, and **SIM** has already completed the simulation since it successfully simulates **REAL** without the honest party's inputs x_n. Based on the hybrid 1 to 11, we can infer that the output of the simulator is computationally indistinguishable from the real output. Completing the proof. $\square$

7.2. Interpretation of results

We summarize PFLM in the active adversary model. Additionally, Table 1 shows the comparison of parameterization between PPML and our scheme for different models, which represents that PFLM has eliminated the additional restricted conditions.

7.2.1. Security against active adversaries

From Theorems 2 and 3, we see that the joint view of any adversarial subset of participants, including the server, can be simulated given no information about the values of the remaining users. This means, no matter how we set the threshold t, users on their own learn nothing about other users, even if the adversarial server collaborates with them. Moreover, Theorem 1 concisely shows the resistance against deceiving attacks in our scheme.

7.2.2. Comparison of parameterization

Table 1 presents the comparison of parameterization between PPML and PFLM in various threat models, where "Minimum threshold" denotes the minimum value of t required for security, "Upper bound of dropouts" is the maximum value of users dropping out allowed by the scheme, and N is the total number of users.

PPML and PFLM are both resistant to deceiving attacks. However, PPML bears strong assumptions described in Table 1. By comparison, PFLM releases the assumptions. Specifically, we reduce "Minimum threshold" from $\lfloor \frac{N}{2} \rfloor + 1$ to 1 and $\lfloor \frac{2N}{3} \rfloor + 1$ to 1 in the Server-only adversary model and Client–Server collusion model, respectively. Additionally, PFLM can tolerate as many users exiting as possible. It deals with up to $N - t$ dropouts rather than $\lceil \frac{N}{2} \rceil - 1$ or $\lceil \frac{N}{3} \rceil - 1$ when the server is adversarial.

Table 1
Comparison of parameterization between PPML and PFLM for different threat models.

Threat model	Minimumthreshold		Upper boundof dropouts	
	PPML	PFLM	PPML	PFLM
Client-only adversary	1	1	$N-t$	$N-t$
Server-only adversary	$\lfloor\frac{N}{2}\rfloor+1$	1	$\lceil\frac{N}{2}\rceil-1$	$N-t$
Client–Server collusion	$\lfloor\frac{2N}{3}\rfloor+1$	1	$\lceil\frac{N}{3}\rceil-1$	$N-t$

8. Performance evaluation

8.1. Implementation of PFLM

The source code for our implementation is available at https://github.com/JiangChSo/PFLM. In this section, we implement PFLM by using python language and the Pypbc library.[2] All experiments are conducted on a desktop with Ubuntu system, an Intel Core i7 CPU, and 32 GB DDR3 of RAM. In the experiments, we simulate a single-threaded cloud server and single-threaded users by using multiple ports. The security level is chosen to be 80 bits for evaluation. For bilinear maps, we choose type A1 pairing and the group order is 160-bit. The group order in result verification is 1024-bit. We adopt the Diffie-Hellman key agreement based on discrete logarithm and the standard Shamir's (t,N)-threshold secret sharing protocol.

Let N denote the number of users and M the number of gradients per user, we evaluate the performance of PFLM in terms of computation and communication. Note that we analyze the performance of membership proof in a separate section (Section 8.4). In our simulation, the used data (i.e., masked gradients) is randomly selected from normal distribution $N(50,20)$. To facilitate cryptographic calculations, we use a parameter L (a magnitude of 10^6) to round the fractional part of chosen numbers, and we can recover each one by dividing L.

In addition, we test PFLM with different proportions of dropouts. We assume that users drop out after sending their shares to all other users, but before sending the masked gradients to the server. This is the worst case since all other users have added the masks of dropped users, and the cloud server must perform an expensive recovery computation to remove them. There is no effect on the performance of PFLM if users drop out at another point in PFLM.

8.2. Performance analysis of client

8.2.1. Computation cost

$O(N^2+MN)$. Each user U_n's computation cost can be broken up into 4 parts: (1) Generating secret shares of N_n^{SK} and β_n takes $O(N^2)$ time, (2) Performing the $N-1$ key agreements and calculating $\mathcal{P}_{n,m}$ for user $U_m \in \mathcal{U}\setminus\{U_n\}$, which take $O(N)$ time, (3) Creating values $\mathbf{PRG}(\beta_n)$, and $\mathbf{PRG}(s_{n,m})$ for other user U_m, which is $O(MN)$ in total since **PRG** stretches the seed to match with the size of users' input vector and (4) In the verification, computing the values for result verification, which takes $O(N)$ time and $O(MN)$ time, respectively. In summary, each user's computation cost is $O(N^2+MN)$.

Fig. 8 demonstrates the running time per user during the verification, which indicates that the running time of verification increases linearly as the number of gradients increases, and increases linearly with the increasing of the number of users when the number is greater than about 70. This is because, the verification cost is mainly related to the number of gradients, and each client must verify each entry of the aggregated result. Besides, some practical operations (such as the generation of variables) affect the running time when the number of users is small, which are ignored gradually as it increases continuously. In addition, Fig. 8 shows more users dropping out results in less time, which keeps a constant as the number of users increases. Fig. 9 shows the total running time of each user, which is a little different from that in Fig. 8. We observe that the total running time increases significantly as the number of users increases. The main reason is that each user must generate secrets shares of N_n^{SK} and β_n for all other users, which takes $O(N^2)$ time.

Fig. 10 presents the comparison between the computation cost of verification and the total cost. In the experiments, we simply consider the proportions of users dropping out are 0% and 30%. We can find the ratio of running time of verification to total running time decreases with the independent variables, including the number of users and gradients. Moreover, the change trend of running time with the growth of users' number is obviously larger than that with the gradient growth, which is consistent with Fig. 9. Since more users dropping out means less validation, the running time of each user decreases with the increasing of the fraction of dropouts, which is described in Figs. 9 and 10.

8.2.2. Communication cost

$O(MN)$. The communication cost of each user U_n can be considered from 5 aspects: (1) Sending 2 and receiving $2(N-1)$ public keys when exchanging the keys with other users, (2) Sending $2(N-1)$ and receiving $2(N-1)$ encrypted shares of N_n^{SK}

[2] Pypbc is a python encapsulation of the PBC library, which provides an implementation of pairing-based cryptography such as bilinear maps, https://github.com/debatem1/pypbc.

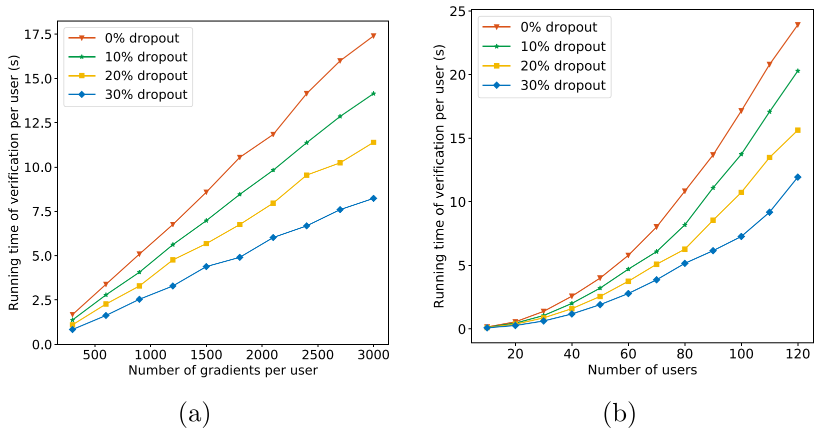


Fig. 8. Total running time per user (Verification). (a) $N = 20$, as the number of gradients per user increases. (b) $M = 100$, as the number of users increases.

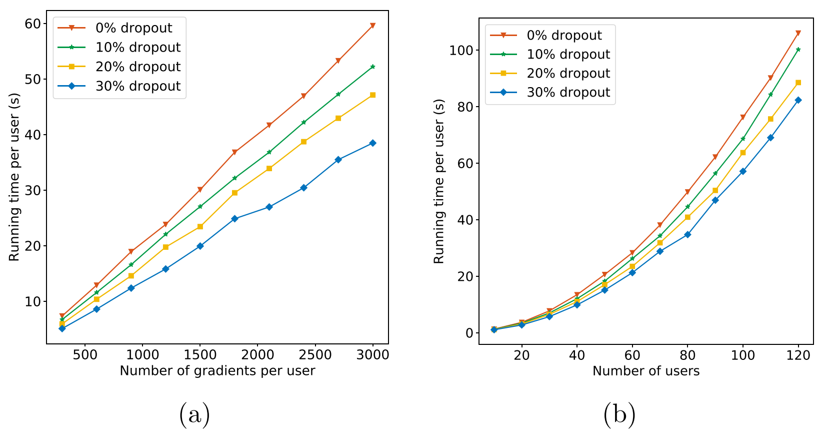


Fig. 9. Total running time per user. (a) $N = 20$, as the number of gradients per user increases. (b) $M = 100$, as the number of users increases.

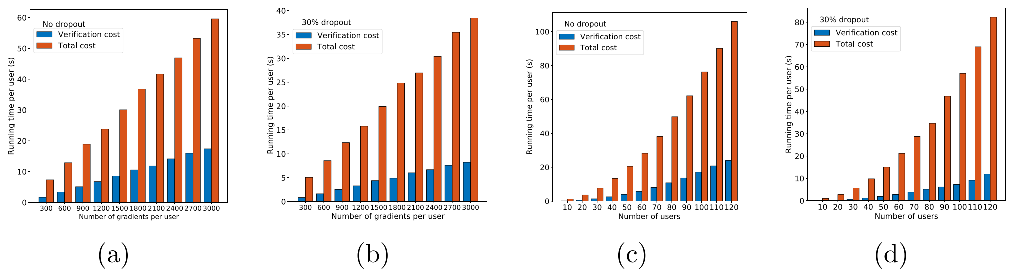


Fig. 10. Comparison between verification computation cost and total cost for each user. (a) No dropout, as the number of gradients per user increases, with $N = 20$. (b) 30% dropout, as the number of gradients per user increases, with $N = 20$. (c) No dropout, as the number of users increases, with $M = 100$. (d) 30% dropout, as the number of users increases, with $M = 100$.

and β_n, (3) Sending the masked data vector of size $M\lceil log_2\mathcal{F}\rceil$ and the messages of size $O(M)$ used in proofs, where the range of inputs is $[0, \mathcal{F}_{\mathcal{N}} - 1]$ and $\mathcal{F} = N(\mathcal{F}_{\mathcal{N}} - 1) + 1$ aiming to avoid overflow, (4) Receiving $N - 1$ masked messages from the server for verifying the correctness of the result later at the beginning of the round **Unmasking**, whose total size is $O(MN)$, (5) Sending n_o secret shares of offline users where $n_o < N - 1$. In summary, the user's communication cost is $O(MN)$.

As seen in Fig. 11, the total transmitted data of any client during the training increases linearly with both the users' number and the gradients' number, and it decreases steadily when more clients drop out. Fig. 12 demonstrates the comparison between communication overhead for verification and total overhead, and the former includes the cost of receiving the data from the cloud server in round **Unmasking** and **Verification**. The main communication cost of any client is for verification,

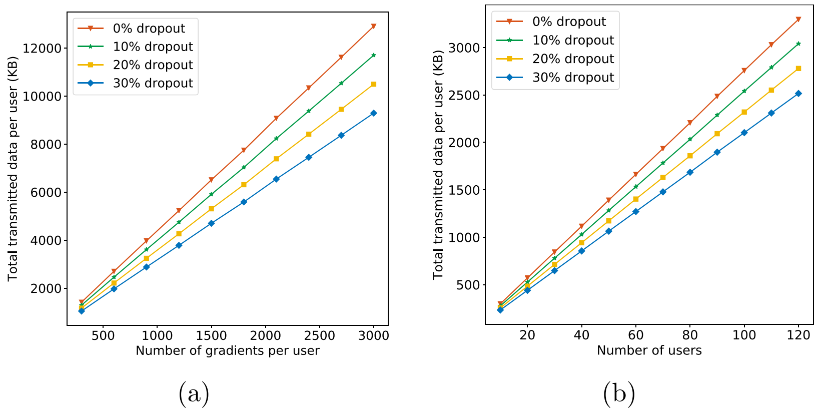


Fig. 11. Total transmitted data per user. (a) $N = 20$, as the number of gradients per user increases. (b) $M = 100$, as the number of users increases.

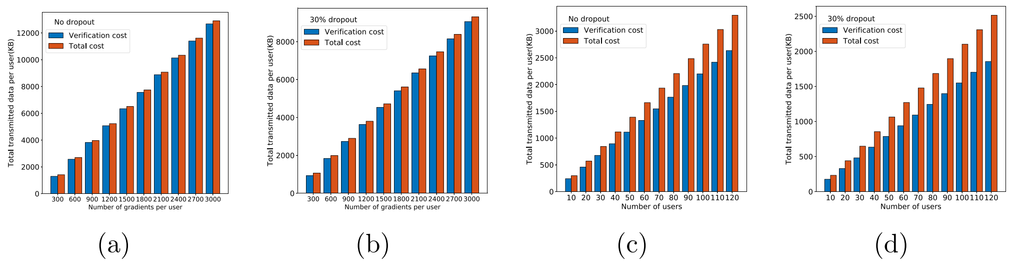


Fig. 12. Comparison between communication cost for verification and total communication cost for each user. (a) No dropout, as the number of gradients per user increases, with $N = 20$. (b) 30% dropout, as the number of gradients per user increases, with $N = 20$. (c) No dropout, as the number of users increases, with $M = 100$. (d) 30% dropout, as the number of users increases, with $M = 100$.

regardless of the number of clients or gradients, and the proportion of the cost for verification does not change significantly when more clients drop out. Furthermore, experiments show that PFLM performs well in terms of communication costs. For example, when the number of users is 20 and the number of gradients in the system is 60000, any user only needs about 12*MB* to complete an update of parameters.

8.3. Performance analysis of server

8.3.1. Computation cost

$O(MN^2)$. The main computation cost of the server is focused on reconstructing the secrets via secret sharing, and generating or removing the appropriate $\mathbf{PRG}(\beta_n)$ and $\mathbf{PRG}(s_{n,m})$ values from the sum $\sum_{U_n \in \mathcal{U}_3} \hat{x}_n$.

(a) Reconstructing N secrets for all users, which takes $O(N^2)$. Each reconstruction $\mathbf{S.recon}(\{(U_n, s_n)\}_{U_n \in \mathcal{M}}, t) \rightarrow s$ can be accomplished via Lagrange polynomials:

$$s = L(0) = \sum_{U_n \in \mathcal{M}} s_n \prod_{U_m \in \mathcal{M} \setminus \{U_n\}} \frac{U_m}{U_m - U_n} \pmod p$$

In round **Unmasking**, the server will receive secret shares of offline users from online users. And the server can calculate the secret value s in two steps. First computing the appropriate Lagrange coefficients

$$\lambda_n = \prod_{U_m \in \mathcal{M} \setminus \{U_n\}} \frac{U_m}{U_m - U_n} \pmod p$$

in $O(N^2)$ time. Then computing $s = \sum_{U_n \in \mathcal{M}} \lambda_n s_n \pmod p$to reconstruct the secret s in $O(N)$ time. Overall, the computation cost of reconstructing all the secrets is $O(N^2)$.

(b) In round **Unmasking**, the server recomputes $\mathbf{PRG}(s_{n,m})$ and $\mathbf{PRG}(\beta_n)$ for user $U_n \in \mathcal{U}_3$ and $U_m \in \mathcal{U}_2 \setminus \mathcal{U}_3$, which take $O(M \cdot |\mathcal{U}_3|)$ and $O(M \cdot |\mathcal{U}_3| \cdot |\mathcal{U}_2 \setminus \mathcal{U}_3|)$ respectively. In the worst case, generating and removing the appropriate values of $\mathbf{PRG}(\beta_n)$ and $\mathbf{PRG}(s_{n,m})$ can take $O(MN^2)$ time in total. Overall, the computation cost of the server is $O(MN^2)$.

Tables 2 and 3 present the computation and communication cost of each round when the number of users and the number of gradients are both 100, where the red font represents the cost of verification, the blue font indicates the cost of receiving users' messages, and the cyan font represents the size of the messages received in the current round. Fig. 13 presents the total running time of the cloud server. Similar to Fig. 9, the server's running time also increases linearly with the increasing of gradients' number, but increases significantly when there are more users. We can see from Fig. 13(a) that when the number of users is low, users dropping out has no significant impact on the running time of reconstructing the secrets and removing the corresponding masks, so that the overall trend of the cost is the same as that of each user. Fig. 13(b) shows that the running time of the server increases with the fraction of dropout, since the server must remove the masks related to the dropped clients after reconstructing the corresponding secrets. And the high cost of dealing with the exited clients is also demonstrated in the server running time in Fig. 2. Besides, we can find the line representing 20% dropout is nearly overlapping with which indicating 30% dropout. From Table 2, we can find the server spends less time in round **MaskedInput** in the case of 20% dropout due to fewer users, while more time in round **Unmasking** since more masks need to be removed, so that the total running time of the server in both cases is almost equal.

8.3.2. Communication cost

$O(N^2 + MN)$. The communication cost of the server relies on the transmission of the encrypted shared secrets, which is $O(N^2)$. Besides, the transmission of masked data vectors and the messages related to the proofs takes $O(MN)$ in total.

Fig. 14 shows the total transmitted data of the cloud server. We can see that the communication cost also increases linearly as the gradients' number grows, and as the number of users grows the costs are very similar even with slight differ-

Table 2
Computation overhead of each round.

	Dropout	Key Sharing	Masked Input	Unmasking	Verification	Total
Client	0%	26327 (ms)	(26870 + 5872) (ms)	3 (ms)	17144 (ms)	76216 (ms)
Client	10%	26106 (ms)	(23109 + 5627) (ms)	3 (ms)	13732 (ms)	68361 (ms)
Client	20%	26210 (ms)	(20861 + 5790) (ms)	3 (ms)	10736 (ms)	63600 (ms)
Client	30%	26653 (ms)	(17743 + 5819) (ms)	3 (ms)	7268 (ms)	59486 (ms)
Server	0%	(25026 + 5219) (ms)	(31521 + 21094) (ms)	(155 + 3545) (ms)	0 (ms)	86560 (ms)
Server	10%	(25050 + 4892) (ms)	(28116 + 17636) (ms)	(242 + 19963) (ms)	0 (ms)	95899 (ms)
Server	20%	(24584 + 2472) (ms)	(26590 + 13988) (ms)	(283 + 32680) (ms)	0 (ms)	100597 (ms)
Server	30%	(24382 + 910) (ms)	(24157 + 11135) (ms)	(69 + 41534) (ms)	0 (ms)	102187 (ms)

Table 3
Communication overhead of each round.

	Dropout	Key Sharing	Masked Input	Unmasking	Verification	Total
Client	0%	232 (KB)	(242 + 22) (KB)	(2177 + 68) (KB)	13 (KB)	2754 (KB)
Client	10%	232 (KB)	(243 + 22) (KB)	(1961 + 68) (KB)	12 (KB)	2538 (KB)
Client	20%	232 (KB)	(243 + 22) (KB)	(1744 + 66) (KB)	11 (KB)	2318 (KB)
Client	30%	232 (KB)	(243 + 22) (KB)	(1527 + 66) (KB)	10 (KB)	2100 (KB)
Server	0%	(22.71 + 23.73) (MB)	(2.2 + 2.13) (MB)	(6.66 + 0.01) (MB)	0 (MB)	57.44 (MB)
Server	10%	(22.71 + 23.74) (MB)	(1.98 + 1.92) (MB)	(6.0 + 0.01) (MB)	0 (MB)	56.36 (MB)
Server	20%	(22.71 + 23.74) (MB)	(1.76 + 1.7) (MB)	(5.17 + 0.01) (MB)	0 (MB)	55.09 (MB)
Server	30%	(22.71 + 23.73) (MB)	(1.54 + 1.49) (MB)	(4.56 + 0.01) (MB)	0 (MB)	54.04 (MB)

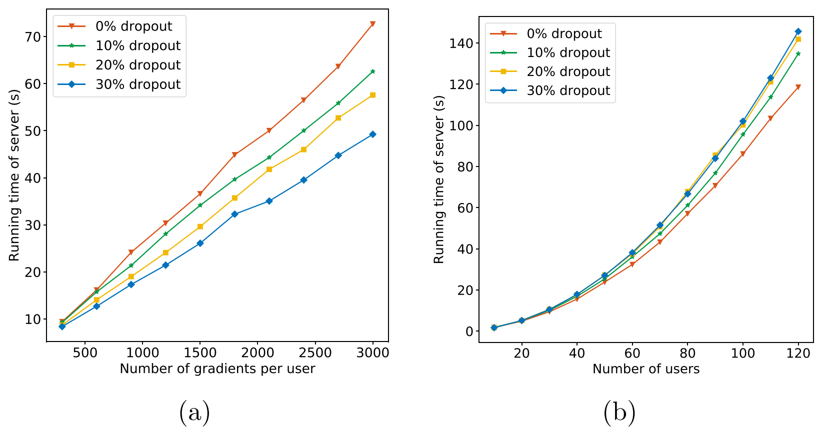


Fig. 13. Total running time of cloud server. (a) $N = 20$, as the number of gradients per user increases. (b) $M = 100$, as the number of users increases.

ences under different proportions of users dropping out. The main reason is that the number of gradients is not numerous enough and the total gradients of exited users account for a small proportion of the total gradients. From Tables 2 and 3, there are other discoveries. For each user, the communication cost is mainly caused by receiving the data from the server. For the cloud server, it needs to receive the messages from all the online clients in all rounds except **Verification**, and the server needs to reconstruct the secrets of the online clients and calculate the aggregated result in **Unmasking** round, which results in large communication and computation costs, respectively.

8.4. Performance analysis of membership proof

In this section, we implement the smart contract in membership proof using Solidity v0.4.23 (more details are in B) and analyze its performance.

8.4.1. Implement of the smart contract

The cloud server issues the membership proofs based on smart contracts. Fig. B.15 shows the $contract_{cs}$ of Tx_1 described in Fig. 6, which mainly consists of two parts *AllProof* and *PartProof*. The functions associated with the two parts are used to write the membership proofs into the storage file and acquire the proofs from blockchain. In specific, the functions *createAllProof* and *createPartProof* will be triggered as the miners generates the new block containing Tx_1. While *createAllProof* is executed, the message $\{h, \eta\}$ is wrote to the storage file, as described in Fig. B.16. While executing *createPartProof*, the message $\{CID, R_{SID}, \{apm_{SID}^{U_n}\}, Rnode_{SID}, apc_{SID}\}$ is stored in its storage file shown in Fig. B.17. Finally, each verifier can acquire the data in the transaction Tx_1 from the public blockchain by triggering *getAllProof* described as Fig. B.18.

8.4.2. Communication and computation cost

In membership proof, the communication costs consist of the time of uploading the transactions to the Ethereum blockchain and the time of getting messages from the blockchain. The latter can be ignored compared with the time of recording the transactions. Hence, the communication cost of membership proof is constant, which is about 15 s. Such a cost can be tolerated from the previous experiments. For the computation overhead, both the server and the users execute a few cryptographic operations mainly consisting of hash functions, which can be ignored compared with the communication cost.

8.5. Performance analysis by comparing with existing schemes

We first estimate the computational costs in terms of basic cryptographic operations, which are shown in Table 4.

The computation costs on the user side of PPML [6], VerifyNet [35], and PFLM are provided in Table 5, where N denotes the number of users, t is the threshold, and N_δ denotes the dropouts. For simplicity, we consider a single gradient per user in the atomic operation analysis. Since users need to verify the membership proofs recorded to Ethereum, they require T_x time to confirm the corresponding transaction. Besides, the proofs in result verification are computed and verified, which requires additional computation costs. However, the above costs protect PFLM from malicious servers. Although VerifyNet and PFLM both achieve verifiability, PFLM does not require users to compute homomorphic hash functions and pseudorandom functions that may incur potential burden in terms of communication and computation costs.

We show a comparison of computation costs on the server side between existing schemes [6,35] and PFLM in Table 6. In PFLM, the computation costs on the user and server are slightly higher than those in PPML and VerifyNet. The additional costs are caused by performing transaction confirmation and membership proof, and make sacrifices in terms of efficiency.

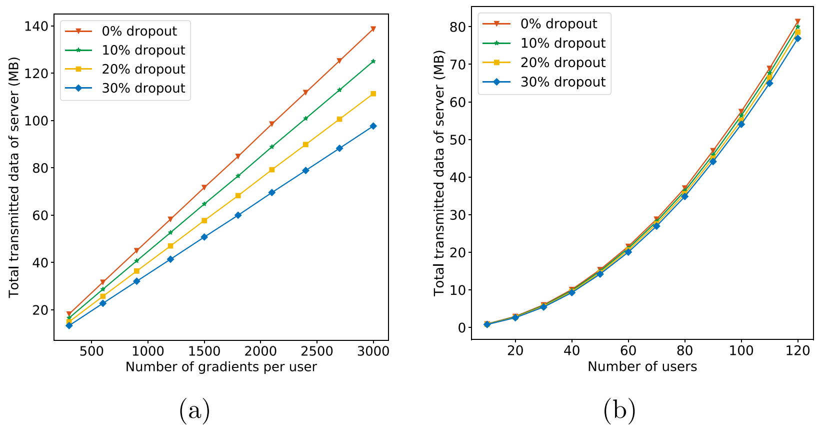


Fig. 14. Total transmitted data of cloud server. (a) $N = 20$, as the number of gradients per user increases. (b) $M = 100$, as the number of users increases.

Table 4
Notation of cryptographic operations.

Symbol	Operation	Symbol	Operation
$Hash_G$	Hash a value into G	$Hash_{Z_p}$	Hash a value into Z_p
Enc	Symmetric-key encryption/decryption	Exp_G	Exponent operation in G
$Pair_{G_T}$	Computing pairing $e(\chi, \xi)$ where $\chi, \xi \in G$	Mul_G	Multiplication in G
T_x	Conducting a transaction in Ethereum	Mul_{Z_p}	Multiplication in Z_p
P_f	Computing a pseudorandom function	Add_{Z_p}	Addition in Z_p
H_f	Computing a homomorphic hash function	Add_G	Addition in G
P_g	Computing a PRG		

Table 5
Computation costs on the user side.

	Computation costs on the user side
PPML [6]	$2N(t-1)\cdot Mul_{Z_p} + 2N(t-1)\cdot Add_{Z_p} + (3N - N_\delta - 1)\cdot Exp_G + 2(N-1)\cdot Exp_{Z_p} + 2(N-1)\cdot Enc + N\cdot P_g$
VerifyNet [35]	$2(Nt - N_\delta)\cdot Mul_{Z_p} + (2Nt - 2N_\delta - 1)\cdot Add_{Z_p} + 4\cdot Exp_G + 2(N-1)\cdot Exp_{Z_p} + 2(N-1)\cdot Enc + N\cdot P_g + 5\cdot Pair_{G_T}$ $+(2N - 2N_\delta + 3)\cdot P_f + 2\cdot Mul_G + H_f$
PFLM	$(2Nt - 2N_\delta - 1)\cdot Mul_{Z_p} + 2N(t-1)\cdot Add_{Z_p} + (N - N_\delta + 5)\cdot Exp_G + (3N - N_\delta + 2)\cdot Exp_{Z_p} + 2(N-1)\cdot Enc + N\cdot P_g$ $+2\cdot Pair_{G_T} + (2N - 2N_\delta + 1)\cdot Hash_G + (N - N_\delta)\cdot Hash_{Z_p} + 4(N - N_\delta)\cdot Add_G + T_x$

Table 6
Computation costs on the server side.

	Computation costs on the server side
PPML [6]	$Nt^2\cdot Mul_{Z_p} + N(t-1)\cdot Add_{Z_p} + (N - N_\delta)N_\delta\cdot Exp_{Z_p} + (N - N_\delta)(N_\delta + 1)\cdot P_g$
VerifyNet [35]	$Nt^2\cdot Mul_{Z_p} + N(t-1)\cdot Add_{Z_p} + (N - N_\delta)N_\delta\cdot Exp_{Z_p} + (N - N_\delta)(N_\delta + 1)\cdot P_g + 4(N - N_\delta - 1)\cdot Mul_G$
PFLM	$Nt^2\cdot Mul_{Z_p} + N(t-1)\cdot Add_{Z_p} + (N - N_\delta)N_\delta\cdot Exp_{Z_p} + (N - N_\delta)(N_\delta + 1)\cdot P_g + Exp_G + T_x$

Table 7
Comparison of communication costs.

	PPML [6]	VerifyNet [35]	PFLM
User	$O(N + M)$	$O(MN)$	$O(MN)$
Cloud server	$O(N^2 + MN)$	$O(N^2 + MN)$	$O(N^2 + MN)$

Table 8
Comparison of security properties.

	PPML	PPDL	SafetyNets	VerifyNet	PFLM
Data Privacy	✓	✓	✗	✓	✓
Robustness to Failures	✓	✗	✗	✓	✓
Verifiability	✗	✗	✓	✓	✓
Resistance against Active Adversaries	✓	✗	✓	✗	✓

However, it surely enables PFLM to secure against deceiving attacks and to provide a stronger security guarantee compared with PPML and VerifyNet. Furthermore, Table 7 presents a comparison of communication costs between existing schemes [6,35] and PFLM, where M denotes the number of gradients per user.

In Table 8, we conduct a security comparison between PFLM and exsisting schemes including PPML [6], PPDL [27], SafetyNets [17], and VerifyNet [35]. Neither PPML and PPDL can support verifiability, and the latter also fails to tolerate users dropping out. Additionally, the problems of data privacy leakage and users dropping out are not considered in SafetyNets, since SafetyNets is mainly designed for verification. On the other hand, PPDL and VerifyNet cannot guarantee the security properties against an actively adversarial server, although they both achieve the confidentiality of data privacy during the execution. By contrast, our scheme is resistant to active adversaries and supports users to verify the aggregated results calculated by the cloud server while guaranteeing the confidentiality of users' local data. Besides, PFLM is supportive of users dropping out during the training.

9. Conclusion

In this paper, we have proposed a privacy-preserving federated learning scheme with membership proof called PFLM. By means of membership proof, PFLM releases the constraint that dropouts must be fewer than half the users in existing practical schemes. Hence, PFLM is more widely appropriate in practice. Additionally, PFLM supports the verifiability of aggregated results returned by the server. Security analysis shows the high security of PFLM in the active adversary model. The implementation of PFLM and experiments demonstrate the performance of PFLM in terms of computation and communication.

CRediT authorship contribution statement

Changsong Jiang: Conceptualization, Writing - original draft, Software. **Chunxiang Xu:** Methodology, Project administration, Writing - review & editing. **Yuan Zhang:** Validation, Investigation, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported in part from the National Nature Science Foundation of China under Grant 61872060, 62002050, Key R&D Program of Sichuan under Grant 2021YFG0158 and the National Key R&D Program of China under Grant 2017YFB0802000.

Appendix A. Proof of correctness

As shown in **Round 4**, the specific verification of the identity-based aggregate signature is as follows, where the equation holds due to the properties of the bilinear map.

$$\begin{aligned}
\hat{e}(S_w, P) &= \hat{e}(\sum_{U_m \in \mathcal{U}_3} S'_m, P) \\
&= \hat{e}(\sum_{U_m \in \mathcal{U}_3} (r_m P_w + sP_{m,0} + c_m s P_{m,1}), P) \\
&= \hat{e}(P_w, \sum_{U_m \in \mathcal{U}_3} r_m P)\hat{e}(\sum_{U_m \in \mathcal{U}_3} P_{m,0} + \sum_{U_m \in \mathcal{U}_3} c_m P_{m,1}, sP) \\
&= \hat{e}(P_w, \sum_{U_m \in \mathcal{U}_3} T_m)\hat{e}(\sum_{U_m \in \mathcal{U}_3} P_{m,0} + \sum_{U_m \in \mathcal{U}_3} c_m P_{m,1}, Q) \\
&= \hat{e}(T_w, P_w)\hat{e}(Q, \sum_{U_m \in \mathcal{U}_3} P_{m,0} + \sum_{U_m \in \mathcal{U}_3} c_m P_{m,1}).
\end{aligned}$$

Each user verifies the equation to ensure that the cloud server aggregates the gradients correctly, which is described as follows.

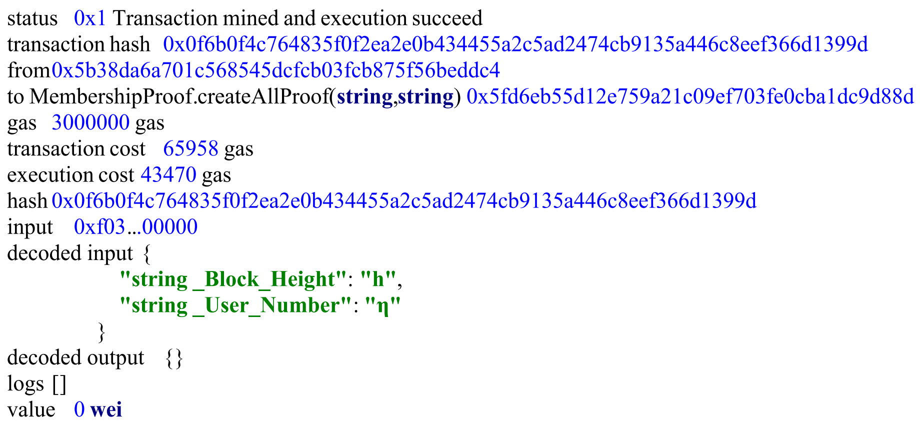
$$\begin{aligned}
z' &= \prod_{U_m \in \mathcal{U}_3} C_m \cdot K^{-sk} \\
&= \prod_{U_m \in \mathcal{U}_3} g^{x_m} \cdot \prod_{U_m \in \mathcal{U}_3} g^{k_m \cdot sk} \cdot K^{-sk} \\
&= \prod_{U_m \in \mathcal{U}_3} g^{x_m} \cdot g^{\sum\limits_{U_m \in \mathcal{U}_3} k_m \cdot sk} \cdot K^{-sk} \\
&= \prod_{U_m \in \mathcal{U}_3} g^{x_m} \cdot g^{\sum\limits_{U_m \in \mathcal{U}_3} k_m \cdot sk} \cdot g^{\sum\limits_{U_m \in \mathcal{U}_3} k_m \cdot (-sk)} \\
&= \prod_{U_m \in \mathcal{U}_3} g^{x_m} = g^{\sum\limits_{U_m \in \mathcal{U}_3} x_m} = g^z.
\end{aligned}$$

Appendix B. Implement of the smart contract

The detailed implementation of the smart contract in membership proof is described as follows.

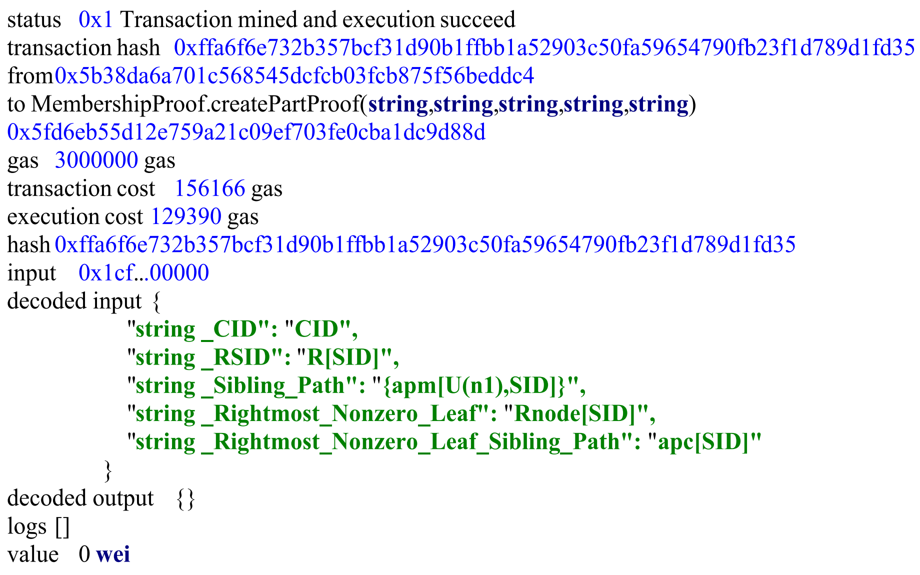
```
pragma solidity ^0.4.23;
pragma experimental ABIEncoderV2;
contract MembershipProof {
   struct PartProof{
      string CID;
      string RSID;
      string Sibling_Path;
      string Rightmost_Nonzero_Leaf;
      string Rightmost_Nonzero_Leaf_Sibling_Path;
   }
   struct AllProof{
      string Block_Height;
      string User_Number;
      PartProof[] Part_Proof;
   }
   PartProof public partProof;
   AllProof public allProof;
   function createPartProof(
      string _CID,
      string _RSID,
      string _Sibling_Path,
      string _Rightmost_Nonzero_Leaf,
      string _Rightmost_Nonzero_Leaf_Sibling_Path
   ) public{
      PartProof memory _partProof = PartProof({
         CID : _CID,
         RSID : _RSID,
         Sibling_Path : _Sibling_Path,
         Rightmost_Nonzero_Leaf : _Rightmost_Nonzero_Leaf,
         Rightmost_Nonzero_Leaf_Sibling_Path : _Rightmost_Nonzero_Leaf_Sibling_Path
      });
      allProof.Part_Proof.push(_partProof);
   }
   function createAllProof(
      string _Block_Height,
      string _User_Number
   ) public {
      allProof.Block_Height = _Block_Height;
      allProof.User_Number = _User_Number;
   }
   function getAllProof() public view returns (AllProof){
      return allProof;
   }
}
```

Fig. B.15. Smart contract called by the cloud server.



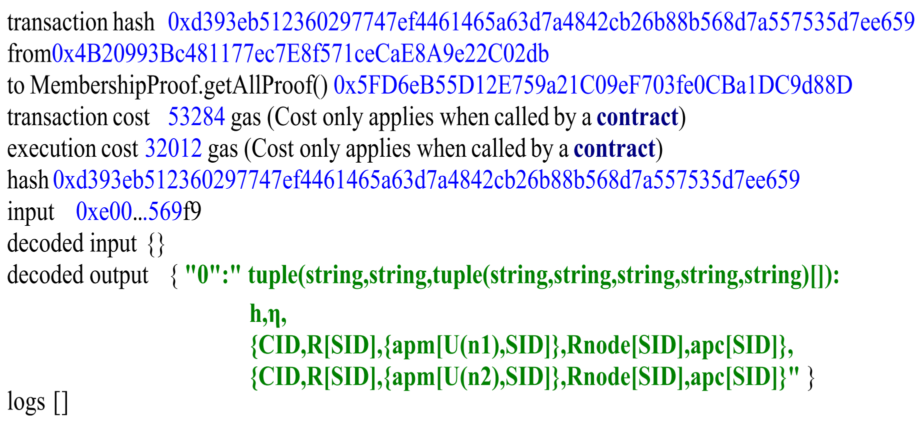
```
status   0x1 Transaction mined and execution succeed
transaction hash  0x0f6b0f4c764835f0f2ea2e0b434455a2c5ad2474cb9135a446c8eef366d1399d
from0x5b38da6a701c568545dcfcb03fcb875f56beddc4
to MembershipProof.createAllProof(string,string) 0x5fd6eb55d12e759a21c09ef703fe0cba1dc9d88d
gas   3000000 gas
transaction cost   65958 gas
execution cost 43470 gas
hash 0x0f6b0f4c764835f0f2ea2e0b434455a2c5ad2474cb9135a446c8eef366d1399d
input    0xf03...00000
decoded input {
                "string _Block_Height": "h",
                "string _User_Number": "η"
             }
decoded output   {}
logs []
value   0 wei
```

Fig. B.16. Allproof written by the cloud server.



```
status   0x1 Transaction mined and execution succeed
transaction hash  0xffa6f6e732b357bcf31d90b1ffbb1a52903c50fa59654790fb23f1d789d1fd35
from0x5b38da6a701c568545dcfcb03fcb875f56beddc4
to MembershipProof.createPartProof(string,string,string,string,string)
0x5fd6eb55d12e759a21c09ef703fe0cba1dc9d88d
gas   3000000 gas
transaction cost   156166 gas
execution cost 129390 gas
hash 0xffa6f6e732b357bcf31d90b1ffbb1a52903c50fa59654790fb23f1d789d1fd35
input    0x1cf...00000
decoded input {
                "string _CID": "CID",
                "string _RSID": "R[SID]",
                "string _Sibling_Path": "{apm[U(n1),SID]}",
                "string _Rightmost_Nonzero_Leaf": "Rnode[SID]",
                "string _Rightmost_Nonzero_Leaf_Sibling_Path": "apc[SID]"
             }
decoded output   {}
logs []
value   0 wei
```

Fig. B.17. Partproof written by the cloud server.



```
transaction hash  0xd393eb512360297747ef4461465a63d7a4842cb26b88b568d7a557535d7ee659
from0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db
to MembershipProof.getAllProof() 0x5FD6eB55D12E759a21C09eF703fe0CBa1DC9d88D
transaction cost   53284 gas (Cost only applies when called by a contract)
execution cost 32012 gas (Cost only applies when called by a contract)
hash 0xd393eb512360297747ef4461465a63d7a4842cb26b88b568d7a557535d7ee659
input    0xe00...569f9
decoded input {}
decoded output   { "0":" tuple(string,string,tuple(string,string,string,string,string)[]):
                    h,η,
                    {CID,R[SID],{apm[U(n1),SID]},Rnode[SID],apc[SID]},
                    {CID,R[SID],{apm[U(n2),SID]},Rnode[SID],apc[SID]}" }
logs []
```

Fig. B.18. Acquiring the proofs.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, Li Zhang, Deep learning with differential privacy, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 308–318..

[2] Michel Abdalla, Mihir Bellare, Phillip Rogaway, The oracle diffie-hellman assumptions and an analysis of dhies, in: Cryptographers' Track at the RSA Conference, 2001, pp. 143–158..

[3] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O Karame, Franck Youssef, Transparent data deduplication in the cloud, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 886–900.

[4] Mihir Bellare, Phillip Rogaway, Random oracles are practical: A paradigm for designing efficient protocols, in: Proceedings of the ACM Conference on Computer and Communications Security, 1993, pp. 62–73..

[5] Manuel Blum, Silvio Micali, How to generate cryptographically strong sequences of pseudorandom bits, SIAM Journal on Computing 13 (4) (1984) 850–864.

[6] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, Karn Seth, Practical secure aggregation for privacy-preserving machine learning, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1175–1191..

[7] Dan Boneh, Matt Franklin, Identity-based encryption from the weil pairing, in: Annual International Cryptology Conference, 2001, pp. 213–229..

[8] Léon Bottou, Large-scale machine learning with stochastic gradient descent, in: Proceedings of COMPSTAT, 2010, pp. 177–186.

[9] Vitalik Buterin et al, A next-generation smart contract and decentralized application platform, White Paper 3 (37) (2014).

[10] Yu. Chen, Fang Luo, Tong Li, Tao Xiang, Zheli Liu, Jin Li, A training-integrity privacy-preserving federated learning scheme with trusted execution environment, Information Sciences 522 (2020) 69–79.

[11] Zijun Cui, Tengfei Song, Yuru Wang, Qiang Ji, Knowledge augmented deep neural networks for joint facial expression and action unit recognition, Advances in Neural Information Processing Systems, 33, 2020..

[12] Whitfield Diffie, Martin Hellman, New directions in cryptography, IEEE Transactions on Information Theory 22 (6) (1976) 644–654.

[13] Taher ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE Transactions on Information Theory 31 (4) (1985) 469–472.

[14] Georg Fuchsbauer, Antoine Plouviez, Yannick Seurin, Blind schnorr signatures and signed elgamal encryption in the algebraic group model, in: Advances in Cryptology – EUROCRYPT, 2020, 2020,, pp. 63–95.

[15] Juan Garay, Aggelos Kiayias, Nikos Leonardos, The bitcoin backbone protocol: Analysis and applications, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2015, pp. 281–310.

[16] Craig Gentry, Zulfikar Ramzan, Identity-based aggregate signatures, in: International Workshop on Public Key Cryptography, 2006, pp. 257–273..

[17] Zahra Ghodsi, Gu. Tianyu, Siddharth Garg, Safetynets: Verifiable execution of deep neural networks on an untrusted cloud, in: Proceedings of the International Conference on Neural Information Processing Systems, 2017, pp. 4675–4684.

[18] Rishab Goyal, Venkata Koppula, Brent Waters, Separating ind-cpa and circular security for unbounded length key cycles, in: IACR International Workshop on Public Key Cryptography, 2017, pp. 232–246.

[19] Briland Hitaj, Giuseppe Ateniese, Fernando Perez-Cruz, Deep models under the gan: Information leakage from collaborative deep learning, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 603–618.

[20] Bargav Jayaraman, Lingxiao Wang, David Evans, Gu. Quanquan, Distributed learning without distress: Privacy-preserving empirical risk minimization, in: Proceedings of the International Conference on Neural Information Processing Systems, 2018, pp. 6346–6357.

[21] Zhuoran Ma, Jianfeng Ma, Yinbin Miao, Ximeng Liu, Privacy-preserving and high-accurate outsourced disease predictor on random forest, Information Sciences 496 (2019) 225–241.

[22] H Brendan McMahan, Eider Moore, Daniel Ramage, Blaise Agüera y Arcas, Federated learning of deep networks using model averaging, arXiv preprint arXiv:1602.05629, 2016..

[23] Yinbin Miao, Jian Weng, Ximeng Liu, Kim-Kwang Raymond Choo, Zhiquan Liu, Hongwei Li, Enabling verifiable multiple keywords search over encrypted cloud data, Information Sciences 465 (2018) 21–37..

[24] Satoshi Nakamoto, A. Bitcoin, A peer-to-peer electronic cash system, Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf, 4, 2008..

[25] Nicolas Papernot, Martín Abadi, Ulfar Erlingsson, Ian Goodfellow, Kunal Talwar, Semi-supervised knowledge transfer for deep learning from private training data, arXiv preprint arXiv:1610.05755, 2016..

[26] NhatHai Phan, Yue Wang, Wu. Xintao, Dejing Dou, Differential privacy preservation for deep auto-encoders: An application of human behavior prediction, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2016, pp. 1309–1316.

[27] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al., Privacy-preserving deep learning via additively homomorphic encryption, IEEE Transactions on Information Forensics and Security 13 (5) (2017) 1333–1345..

[28] Ahmed Salem, Yang Zhang, Mathias Humbert, Mario Fritz, Michael Backes, Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models, in: Network and Distributed Systems Security Symposium, 2019.

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

[30] Reza Shokri, Vitaly Shmatikov, Privacy-preserving deep learning, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 1310–1321.

[31] Reza Shokri, Marco Stronati, Congzheng Song, Vitaly Shmatikov, Membership inference attacks against machine learning models, in: IEEE Symposium on Security and Privacy, 2017, pp. 3–18.

[32] Florian Tramer, Dan Boneh, Slalom: Fast, verifiable and private execution of neural networks in trusted hardware, in: International Conference on Learning Representations, 2018.

[33] Fengwei Wang, Hui Zhu, Lu. Rongxing, Yandong Zheng, Hui Li, A privacy-preserving and non-interactive federated learning scheme for regression training with gradient descent, Information Sciences 552 (2021) 183–200.

[34] Gavin Wood et al, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Paper 151 (2014) (2014) 1–32.

[35] Xu. Guowen, Hongwei Li, Sen Liu, Kan Yang, Xiaodong Lin, Verifynet: Secure and verifiable federated learning, IEEE Transactions on Information Forensics and Security 15 (2019) 911–926.

[36] Yu. Jun, Zhenzhong Kuang, Baopeng Zhang, Wei Zhang, Dan Lin, Jianping Fan, Leveraging content sensitiveness and user trustworthiness to recommend fine-grained privacy settings for social image sharing, IEEE Transactions on Information Forensics and Security 13 (5) (2018) 1317–1332.

[37] Yu. Jun, Baopeng Zhang, Zhengzhong Kuang, Dan Lin, Jianping Fan, Iprivacy: Image privacy protection by identifying sensitive objects via deep multi-task learning, IEEE Transactions on Information Forensics and Security 12 (5) (2016) 1005–1016.

[38] Lei Yu, Ling Liu, Calton Pu, Mehmet Emre Gursoy, Stacey Truex, Differentially private model publishing for deep learning, in: IEEE Symposium on Security and Privacy, 2019, pp. 332–349..

[39] Pengfei Zhang, Cuiling Lan, Wenjun Zeng, Junliang Xing, Jianru Xue, Nanning Zheng, Semantics-guided neural networks for efficient skeleton-based human action recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2020, pp. 1112–1121.

[40] Xiaojun Zhang, Xu. Chunxiang, Huaxiong Wang, Yuan Zhang, Shixiong Wang, Fs-peks: Lattice-based forward secure public-key encryption with keyword search for cloud-assisted industrial internet of things, IEEE Transactions on Dependable and Secure Computing (2019), https://doi.org/10.1109/TDSC.2019.2914117.

[41] Yuan Zhang, Xu. Chunxiang, Nan Cheng, Hongwei Li, Haomiao Yang, Xuemin Shen, Chronos+: An accurate blockchain-based time-stamping scheme for cloud storage, IEEE Transactions on Services Computing 13 (2) (2019) 216–229.

[42] Yuan Zhang, Chunxiang Xu, Nan Cheng, Xuemin Sherman Shen, Secure password-protected encryption key for deduplicated cloud storage systems, IEEE Transactions on Dependable and Secure Computing (2021), https://doi.org/10.1109/TDSC.2021.3074146.
[43] Yuan Zhang, Chunxiang Xu, Hongwei Li, Kan Yang, Nan Cheng, Xuemin Sherman Shen, Protect: Efficient password-based threshold single-sign-on authentication for mobile users against perpetual leakage, IEEE Transactions on Mobile Computing, 2020, doi: 10.1109/TMC.2020.2975792..
[44] Yuan Zhang, Chunxiang Xu, Xiaodong Lin, Xuemin Sherman Shen, Blockchain-based public integrity verification for cloud storage against procrastinating auditors, IEEE Transactions on Cloud Computing (2019), doi: 10.1109/TCC.2019.2908400..
[45] Yuan Zhang, Chunxiang Xu, Jianbing Ni, Hongwei Li, Xuemin Sherman Shen, Blockchain-assisted public-key encryption with keyword search against keyword guessing attacks for cloud storage, IEEE Transactions on Cloud Computing (2019), doi: 10.1109/TCC.2019.2923222..