2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3315-1596-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICPADS63350.2024.00034

FLoomChecker: Repelling Free-riders in Federated Learning via Training Integrity Verification

Guanghao Liang*, Shan Chang*, Minghui Dai*, Hongzi Zhu†
*School of Computer Science and Technology, Donghua University, Shanghai, China
†Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai, China
guanghao@mail.dhu.edu.cn, changshan@dhu.edu.cn, minghuidai@dhu.edu.cn, hongzi@cs.sjtu.edu.cn

***Abstract*—Federated learning is a mechanism that allows participating clients to train locally with their own data in order to receive rewards, thus avoiding the transfer of data to a central server and protecting users' privacy. However, some "lazy" clients may adopt the strategy of fabricating false model local updates in an attempt to "free-riding" without actually contributing real data or consuming local computational resources. To address this issue, we propose FLoomChecker, an integrity detection scheme for federated learning training models. The scheme combines the techniques of trusted execution environments and Bloom filters to efficiently identify clients that do not train honestly by committing and proving. We conducted experimental evaluations of FLoomChecker, examining three main aspects: query time, build time, and memory footprint in trusted execution environment (TEE). The experimental results demonstrate the effectiveness of our scheme, and its performance improves as the number of local training rounds increases.**

***Index Terms*—Federated Learning, Free-rider attack, TEE, Bloom filter, Training integrity verification.**

I. Introduction

Federated Learning (FL), a distributed learning framework proposed by Google, allows clients to keep data on their respective local devices, enabling them to train models without sharing raw data. Therefore, the emergence of FL provides a new idea and solution to address the privacy leakage and data security issues brought about by centralized data training, facilitates data sharing and collaboration among various clients, promotes the flow and application of data among multiple parties, improves the utilization value and efficiency of data, and alleviates to some extent the contradiction between the protection of data privacy and the enhancement of model training accuracy. To fully realize the potential of FL, some of these common security issues need to be identified and addressed first [1]. Particularly in specific scenarios such as healthcare data sharing and financial transaction data analysis, clients may not be completely trustworthy, which may lead to various risks. For example, some clients may engage in free-rider attacks (FRA). FRA is a common local model manipulation attack. It refers to the fact that since the local model training process is done locally by clients, some "lazy" or "selfish" clients, do not faithfully follow the predefined protocols of the FL framework to complete the local training process, but submit fabricated false model local updates, with the intention of not contributing real data or consuming local computational resources to the FL framework [2]. This not only destroys the fairness among clients, but also reduces the training efficiency of the federated model and affects the accuracy of the global model.

We emphasize that free-riders are often not "malicious" but "selfish", and their goals are usually not to disrupt federation training, but to obtain something for nothing (or to get excessive revenue). Although both free-riders and malicious clients submit anomalous local updates, detecting free-riders is more difficult than detecting malicious clients. This is because the anomalous behaviour of free-riders is more subtle or mild. For example, a "free-rider" might fake new local updates by making small perturbations to model updates from the previous round of training or global model updates from the latest round. The local updates generated in these ways are not particularly outrageous. Traditional anomaly detection models have two approaches: they either find anomalies or tolerate them. However, for free-riders, it is very difficult to detect their anomalous behaviour because the errors in these models are not particularly obvious. Tolerating such behaviour is also infeasible because all clients can be considered selfish compared to a few malicious clients. Without a reasonable checking mechanism, everyone can engage in faking at low cost, which ultimately leads to a decrease in the overall performance and trustworthiness of the model. Therefore, in order to eliminate the free-riding behaviour of the selfish, a training integrity check needs to be performed on each submitted update, which means checking that the client faithfully completes the local training with local data.

One possible idea is to use trusted execution environment (TEE) for model integrity checking [3], [4]. The idea is as follows. Each client holds a pair of keys with the server, and the key used for decryption at the server side is stored in the TEE and can only be accessed by the TEE. After local training, the client uploads the local training results and the encrypted original training data to the server for training integrity checking. The server will use the key to decrypt the original training data in the TEE and reproduce the local training in the TEE, and then compare the training results with the results provided by the client. If the two results are the same, then the training integrity check passes. Otherwise, it fails, which means that the local training process of the client is not legitimate, and the corresponding local training updates will be discarded. Since the original training data

Shan Chang is corresponding author.

979-8-3315-1596-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ICPADS63350.2024.00034

and the training execution process are stored and completed in the TEE during the whole process, its confidentiality can be guaranteed. However, performing these operations on the server side consumes a lot of computing resources and storage space.

TrustFL proposed by Zhang*et al.* solves this problem. TrustFL requires the client to support TEE locally. When the client completes the training locally, it needs to submit a commitment message to the local TEE indicating that all the training rounds have been completed truthfully. Subsequently, the local TEE selects a small number of training rounds for validation, thus avoiding uploading all the information to the server for checking [5]. In TrustFL, Merkle trees are used to check for commitments. However, as the number of local training rounds increases, the time to build the Merkle tree and query whether a node is in the tree increases significantly.

We propose an approach based on dual Bloom filters, called FLoomChecker, which still uses the "commit and prove" scheme and requires the client to support TEE locally. By using two Bloom filters, we achieve storage and efficient checking of promises. Although there is a trade-off between memory footprint and false positive rate, Bloom filters are very fast to build and find since they are essentially binary arrays [6], [7]. In addition, to prevent the client from reusing old local updates to fake the execution of the entire training process, we use dynamic inputs to ensure that each round of local training is performed on a new dataset, thus guaranteeing the freshness of training results for the client.

We implement our approach on Lenet5 model training and MLP models and compare it with TrustFL. The experimental results show that FLoomChecker reduces the query time by 65.82% compared to TrustFL under the condition of 1000 rounds of local training and 0.01 false positive rate, and FLoomChecker reduces the build time by 17.46% compared to TrustFL under 1000 rounds of local training and 0.1 false positive rate. Moreover, the advantage of FLoomChecker extends further as the number of local training rounds increases. We found experimentally that the lower the false positive rate, the more memory space is required and the slower it is, but the memory footprint and processing speed of verifying integrity using our method is far superior to storing the information directly into TEE.

II. Related Work

Now there are two kinds related to our work, including malicious attack detection and Byzantine robust defence these approaches focus on defending against malicious attackers.

Malicious Attack Detection. Liu *et al.* propose an isolation forest (iforest)-based malicious model detection mechanism, named D2MIF, which can effectively detect malicious models and significantly improve the accuracy of global models [8]. Yazdinejad *et al.* introduce an internal auditor for evaluating the similarity and distribution of encrypted gradients to distinguish between benign and malicious gradients by using a Gaussian mixture model and the Mahalanobis Distance for Byzantine tolerance aggregation [9]. Jeong *et al.* proposed that FedCC identifies and filters malicious clients by leveraging Centered Kernel Alignment (CKA) similarity [10]. However, it is often difficult for these detection strategies to detect "mild" free-rider models because the differences between these models and the benign models are extremely small.

Byzantine Robust Defence. Existing Federated Learning Byzantine Robust Defence approaches focus on eliminating or weakening "anomalous" local updates. For example, Trimmed Mean [11] reduces the interference of anomalies in the model by removing the highest and lowest portions of the update values to compute the mean. Zhang *et al.* proposed a novel defense algorithm based on the pivot learning method, called federated adversarial training (FAT), which aims to improve the robustness of the conventional FL protocol [12]. FRL [13] utilises Supermask Training and edge popping algorithms to reduce the impact of outliers by making the acceptable update space sparser and cleaner. ToFi [14] effectively defends against Byzantine attacks in FL by using a reference dataset on the server-side to evaluate and filter out anomalous computational results and combining it with an α-weighted averaging approach to aggregate and update the global model based on the loss of information uploaded by each worker node. However, such methods only work effectively in the presence of a small number of malicious clients. Once the majority of clients exhibit selfish behaviour, the fairness of the system is significantly undermined.

III. Models

A. System Model

We consider an FL system consists of a central server, denoted by $\mathcal{S}$, and N clients that have relevant data and support TEE. Moreover, each client c_k shares a pair of public and private keys $(PRIV_k, PUB_k)$ with the server. We assume that a Public Key Infrastructure (PKI) is used to distribute these keys, where the private key $PRIV_k$ is stored inside the TEE of c_k and the public key PUB_k is stored on $\mathcal{S}$. The whole process is as follows.

1) Task definition and release. $\mathcal{S}$ explicitly defines the goal of the task, the training algorithm used on the platform, and sets the relevant parameters. $\mathcal{S}$ initializes a global model w_0 before training starts.

2) Client selection and local training. $\mathcal{S}$ randomly selects K clients from these N clients to participate in the latest round of global model training with a certain sampling probability. The selected client c_k trains the model on local data $\mathcal{D}_k$ and updates the model parameters using the gradient descent. Let the loss function be denoted by $\mathcal{L}(w; \mathcal{D}_k)$, then the model update formula for c_k in the t round of iterations is expressed as

$$w_k^{(t+1)} = w_k^{(t)} - \eta \nabla \mathcal{L}(w_k^{(t)}; \mathcal{D}_k). \tag{1}$$

where η is the learning rate.

3) Model uploading. Each client uploads its model update $\Delta w_k^{(t+1)} = w_k^{(t+1)} - w_k^{(t)}$ to $\mathcal{S}$ after local training.

4) Global aggregation. $\mathcal{S}$ receives model updates from all clients and performs a weighted average to update the global model. The update formula is expressed as

$$w^{(t+1)} = w^{(t)} + \frac{1}{K}\sum_{k=1}^{K}\Delta w_k^{(t+1)}. \tag{2}$$

5) Iteration. Repeat step 2 to step 4 until the model converges or reaches a predetermined number of training rounds.

B. Attack Model

We assume that the server is honest but curious, and that it meticulously coordinates the training process for FL, making sure that all steps are on track. However, the server also holds a great interest in the client's sensitive data, always wants to explore and learn about it.

Clients are selfish and motivated to attack for unearned benefits. They want to maximise their benefits without spending a penny or investing as few resources as possible. They may 1) bypass the local training process and directly generate random model updates without using real data or training process. 2) try to reduce the consumption of computational resources by modifying the parameters of the model or freezing some layers. 3) record the correct model updates in the first few rounds of normal training and then repeatedly submit these histories as their training results in the subsequent rounds. 4) fake new local updates by introducing small perturbations to model updates from previous training rounds or the latest global model update [15]. In addition, sampling decisions are seen when the client looks at the training algorithms published by the server to ensure that its own training data is not stolen. The exposed sampling decisions may allow the client to evade detection by training only the rounds that need to be detected. Note that unlike the previous assumption that the number of attacking clients should be less than half of the total number of clients [9]–[11], [13], [14], we now assume that all clients are potentially selfish.

C. Design Goals

Our goal is to design a scheme to achieve the following two aspects. (1) Model training integrity verification: to ensure that the model updates submitted by each client are based on real data training to prevent any form of manipulation or tampering. Our scheme is effective in detecting clients that submit fake updates without actually performing training. (2) Privacy protection: during the validation process, we ensure that the client's training data and model updates are not leaked, safeguarding the privacy and security of each client.

IV. "Commit and Prove" Through TEE

Commit outside TEE. To prevent sampling decisions from being exposed, each client stores a hash summary of all model parameters and makes a commitment after completing local training. They commit to complete each round of training faithfully and ensure that the results of each round can pass verification. It is worth mentioning that the reason why we store the hash values of the model parameters instead of storing the original model parameters is that storing the former requires much less memory space than storing the latter, which can significantly reduce the memory usage of TEE.

Prove inside TEE. TEE calls the model parameters w^i for round i, w^{i+1} for round $i+1$ and the corresponding dataset $\mathcal{D}^i$. Throughout the process, TEE ensures that sampling and subsequent operations are carried out in strict accordance with the programme, without interference from the client. It verifies the consistency of w^i with the commitment information. Then it reproduces the training process using w^i and $\mathcal{D}^i$ and compares the reproduced results with w^{i+1}, and if they are consistent, it proves that the client's model training process has integrity.

V. Design of FLoomChecker

A. Overview

The server pre-sets the false positive rate F of the dual Bloom filters. Due to the layered relationship between BFA and BFB, both can take $\sqrt{F}$ as their false positive rate p. At the end of each training round, the client stores the model parameters for that round and performs a hash operation on these parameters. It is worth mentioning that the reason why we use the hash values of the model parameters for comparison instead of using the original model parameters directly is that the memory space required for the hash values is much smaller than that required for storing the original model parameters, which can greatly reduce the memory usage of TEE. After local training, a procedure is called within TEE to build two Bloom filters. First, the program builds Bloom filters based on the false positive rate and the number of local training rounds. TEE requests the hash values of the model parameters in batches and stores these hash values accurately in the two Bloom filters. The process of commitment generation is shown in Fig. 1.

During validation, TEE calls a procedure to perform several rounds of random checks. The first step of the check is that the programme calls the model parameters of the selected round w_i and the hash summary a_{i+1} of the model parameters of the $i+1$ round, and after hashing w_i, check whether the obtained hash results a_i and a_{i+1} are in the two Bloom filters. Then, the procedure reproduces the training process using the dataset $\mathcal{D}^i$ for that round. Finally, the program hashes the reproduced results and verifies that they are consistent with a_{i+1}. The process of commitment verification is shown in Fig. 2.

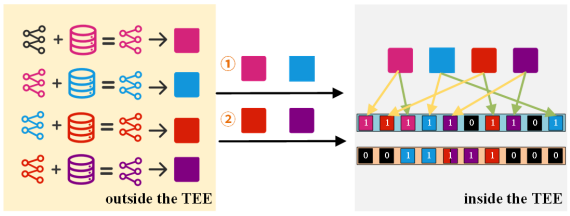

Fig. 1: Commitment Generation.

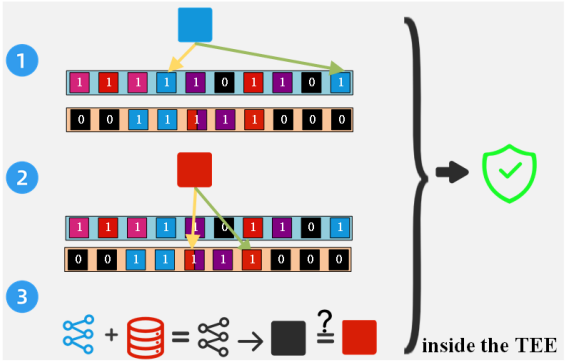

Fig. 2: Commitment Verification.

B. Commitment Checking Based on Dual Bloom Filters

A straightforward commitment scheme is to put the model parameter hashes of all rounds into the TEE, and then randomly select a number of pairs of consecutive model parameter hashes to check [5]. However, this approach consumes much of the TEE's memory, as the memory required to put all these model parameter hashes into the TEE increases with the total number of local training rounds E. Therefore, we propose a Bloom filter-based commitment checking method.

The Bloom filter is a binary array that requires far less memory than that of directly storing hashes of all model parameters. Although the Bloom filter has a certain false positive rate, where a summary of the hash values of the model parameters does not actually exist but is mistakenly assumed to exist, we can accept this method due to the small false positive rate.

The steps of commitment checking based on dual Bloom filters are as follows.

1) After the local training is completed, the client will hash the model parameters for each round. The calculation formula is shown in Eq. (3). $NNL(.)$ denotes the neural network learning, w_i denotes the model parameters of the i-th round, and $\mathcal{D}^i$ denotes the dataset used for the i-th round of training. $H(.)$ denotes the hash operation on the resulting data, and a_{i+1} denotes the hash digest of the model parameters of the $i+1$th round.

$$\mathrm{a}_{i+1} = H\left(NNL\left(w_i, \mathcal{D}^i\right)\right). \tag{3}$$

2) TEE builds two Bloom filters BFA, BFB using the false positive rate p and the total number of local training rounds E. In addition, the number of hash functions ξ can be computed using Eq. (4), and the bit array size m can be computed using Eq. (5). Eq. (4) and Eq. (5) ensure that the Bloom filter operates with a desirable false positive rate and a sufficient bit array size [6].

$$\xi = \log_2 \frac{1}{p}. \tag{4}$$

$$m = \frac{-E \times \ln p}{(\ln 2)^2}. \tag{5}$$

3) TEE requests hashes of model parameters in batches. For each model parameter's hash value a_i, it is first processed using the Bloom filter's hash functions, then the hash result is taken modulo the array length m, obtaining the index IA_i in the BFA bit array, and the corresponding position in the BFA bit array is set to 1. Next, when inserting the data into BFB, IA_i is first added to the previous round's index IA_{i-1}, then the sum is taken modulo the array length m, obtaining the index in the bit array, and the corresponding position in the BFB bit array is set to 1. The details are illustrated in Algorithm 1.

Algorithm 1 FLoomChecker: Element Insertion in Bloom Filter

Input: Hash values for a batch $B=\{\mathrm{a}_{o_1}, \mathrm{a}_{o_2}, \ldots, \mathrm{a}_{o_L}\}$**, False positive rate of dual Bloom filters** F**, Hash functions in Bloom filters** $f_1, f_2, \ldots, f_\xi$**, Bloom filter array length** m

Output: Bloom filter BFA**, Bloom filter** BFB

$BFA = Bloom\_Filter\_Initialization(m, \sqrt{F})$
$BFB = Bloom\_Filter\_Initialization(m, \sqrt{F})$
for $i = o_1, o_2, \ldots, o_L$ **do**
  $HA_i = [f_1(\mathrm{a}_i), f_2(\mathrm{a}_i), \ldots, f_\xi(\mathrm{a}_i)]$
  //Hashing elements using the Bloom filter's hash function
  $IA_i = HA_i \ mod \ m$
  $SET(BFA, IA_i) = 1$
  //Set the position indicated by IA in the BFA to 1
  $IB_i = (IA_i \ + \ IA_{i-1}) \ mod \ m$
  $SET(BFB, IB_i) = 1$
end for
return BFA, BFB

4) A series of verification rounds are randomly generated in the TEE, which can be expressed as

$$Random\ selection\left(\mathrm{s}_1,\ \mathrm{s}_2 \ldots, \mathrm{s}_Q\right). \tag{6}$$

where Q denotes the total number of rounds generated in TEE.

5) In order to verify that the s_i-th round performed the training honestly, the TEE requests the s_i-th round model parameter w_{si}, the hash value $\mathrm{a}_{\mathrm{si}+1}$ of the model parameter for the s_{i+1}-th round, and the corresponding dataset $\mathcal{D}^{\mathrm{si}}$. The first step is to verify that these parameters are consistent with the commitment, i.e., the hash function of the Bloom filter hash $H(w_{\mathrm{si}})$ and the hashed result is taken modulo the length of the array m to obtain the positional index IA, and to verify that the positions of IA in the BFA bit array are all 1. The hash function of the Bloom filter then hash $\mathrm{a}_{\mathrm{si}+1}$ and the hashed result is taken modulo the length of the array m to obtain the positional index I. Verify that the positions of I+IA in the BFB bit array are all 1. After verifying the commitment, use w_{si} and $\mathcal{D}^{\mathrm{si}}$ to reproduce the training process of the s_i rounds and compare the reproduced results with $\mathrm{a}_{\mathrm{si}+1}$ to check for consistency.

6) The TEE calling program generates the proof $\mathrm{SIG_c}$ based on Eq. (7). The process of generating proofs inside the TEE is shown in Algorithm 2.

$$S\left(PRIV, \mathrm{a}_0^t, \Psi_\mathrm{c}^t\right) \rightarrow \mathrm{SIG_c}. \tag{7}$$

where $S(.)$ is the signature function, $PRIV$ denotes the private key, a_0^t is the latest global model hash summary, and Ψ_c^t denotes the updated gradient of the model for client c.

7) The server is validated according to Eq. (8), where $V(.)$ is the verification function and PUB is the public key. If the result is True, it waits to aggregate that update while the server issues a task reward to client c. Otherwise, the gradient and the client will be rejected.

$$V\left(PUB, \mathrm{a}_0^t, \Psi_\mathrm{c}^t, \mathrm{SIG_c}\right) \rightarrow True/False. \tag{8}$$

Algorithm 2 FLoomChecker: Proof Generation inside TEE

Input: Selected rounds $\mathrm{s}_1,\ \mathrm{s}_2 \ldots, \mathrm{s}_Q$**, Dynamic seed** DS**, Batch size** SEQ**, Bloom filter** BFA**, Bloom filter** BFB **, Hash functions in Bloom filters** $f_1, f_2, \ldots, f_\xi$**, Bloom filter array length** m **, Hash value for the latest round of global models** a_0

Output: Proof SIG**, Local update** Ψ

for $i = \mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_Q$ **do**
  $w_\mathrm{i} = requesting(i)$
  $\mathrm{a_{i+1}} = requesting\_hash(i+1)$ //Request for model parameters for round i and hash value for round i+1
  $\mathrm{a_i} = H(w_i)$
  $HA_i = [f_1(\mathrm{a}_i), f_2(\mathrm{a}_i), \ldots, f_\xi(\mathrm{a}_i)]$ //Hashing elements using the Bloom filter's hash function
  $IA_i = HA_i \ mod \ m$
  $HA_{i+1} = [f_1(\mathrm{a}_{i+1}), f_2(\mathrm{a}_{i+1}), \ldots, f_\xi(\mathrm{a}_{i+1})]$
  $I = HA_{i+1} \ mod \ m$
  $IB_i = (IA_i \ + \ I) \ mod \ m$
  $Check(BFA, IA_i) = 1 \ and \ Check(BFB, IB_i) = 1$ //Validation is passed if the position indicated by IA_i in BFA and the position indicated by IB_i in BFB are both set to 1
  $\mathcal{D}_i = use(SEQ, DS)$ //obtain the dataset for this round using SEQ and DS
  $w_{i+1}^{wait} = NNL(w_i, \mathcal{D}_i)$
  $\mathrm{a}_{i+1}^{wait} = H(w_{i+1}^{wait})$
  $Check\_equal(\mathrm{a}_{i+1}^{wait}, \mathrm{a}_{i+1})$
end for
$w_\mathrm{E} = requesting(E)$
$w_0 = requesting(0)$
$\Psi = w_\mathrm{E} - w_0$
$S\left(PRIV, \mathrm{a}_0, \Psi\right) \rightarrow \mathrm{SIG}$
return SIG, Ψ

C. Ensuring The Freshness of Model Training Results

In FL, it is important to keep the freshness of model training results since the data is updated and trained independently on each client. If freshness is not guaranteed, an attacker may repeatedly submit model training results for several rounds to evade detection, or keep submitting correct model training results to repeatedly obtain rewards.

A common solution idea is to ensure the freshness of client-side training results through dynamic inputs that ensure that each round of training is performed on a new randomized dataset. A simple approach is to pass all the training datasets inside the TEE, and use the same function inside and outside the TEE to extract the training data of the i-th round [16]. However, TEE usually has limited memory space, which may lead to performance degradation if too much data is stored. This is because a large amount of data may require more computational resources to process, which in turn affects the operation of other tasks. In addition, if the amount of data exceeds the memory capacity of the TEE, it may lead to memory overflow, which may trigger program crashes or unpredictable behaviors.

To solve these problems, we propose a unique strategy. The specific steps are as follows.

1) Processing of data. The TEE divides the data into d different batches. And indexes and Hash-based Message Authentication Code (HMAC) [17] are generated for each batch. these data are recovered outside TEE before starting local training.

2) Request for dynamic seed and batch count. Before starting local training, the client sends a request to the server for a dynamic seed (DS) and a batch count (SEQ), where 1 $\leq$ SEQ $\leq$ 7 and is an integer. DS and SEQ are randomly generated by the server to ensure their uniqueness and unpredictability.

3) Determining the training batch. In each round of training, the client selects the training batch based on DS and SEQ. Specifically, the dataset $\mathcal{D}^i$ trained in the i-th round consists of SEQ $\mathcal{D}_j^i$, where $\mathcal{D}_j^i$ is determined by Eq. (9).

$$\mathcal{D}_j^i = (DS \times i + j) \bmod d, \ j \in \{0, 1, \ldots, SEQ\}. \tag{9}$$

This ensures that each round of training is performed on a new random dataset.

4) Validation inside TEE. When the TEE requests validation, the DS and SEQ information can be used to find the corresponding $\mathcal{D}^i$ and check the integrity of the training batch using HMAC, and finally reproduce the training process to determine the integrity of the model training.

Our proposed strategy enhances security through dynamic inputs, but this is not enough. We also need to ensure that the global model used is from the latest round. Therefore, we bind DS and SEQ to the hash summary a_0^t of the latest round of the global model and include a_0^t as part of the proof. This ensures that each round of training is based on the latest global model, which further improves the freshness and security of the training results.

VI. Experiments

A. Experiment Setup

We conduct a series of experiments to validate the performance of FLoomChecker, all of which are implemented using

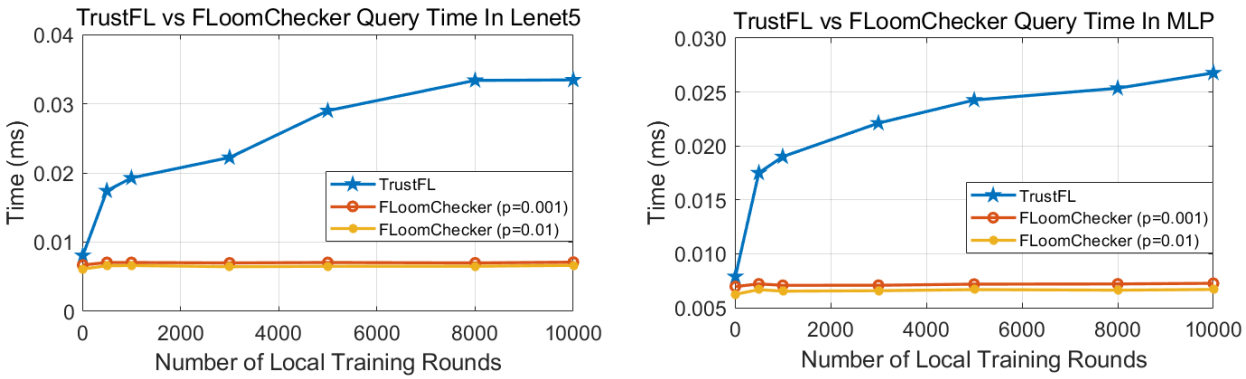

Fig. 3: Merkle Tree vs. Bloom Filter Query Time.

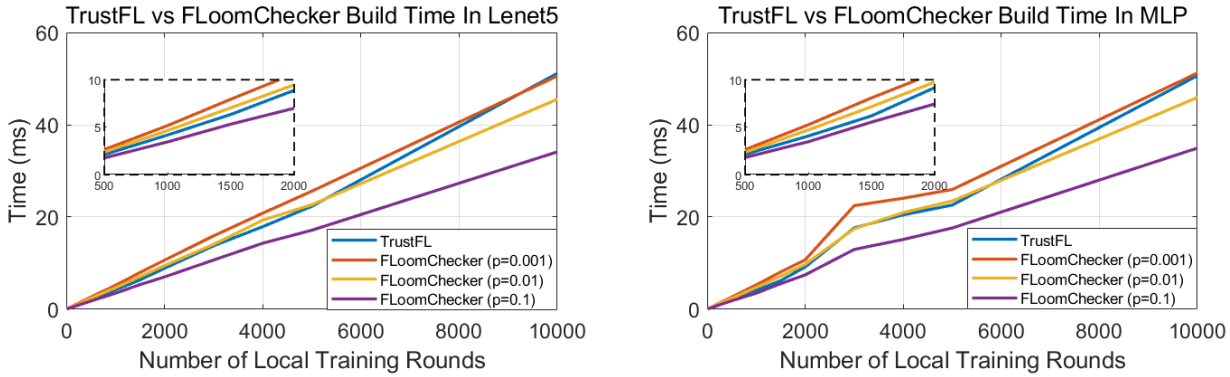

Fig. 4: Merkle Tree vs. Bloom Filter build Time.

PyTorch and Intel(R) UHD Graphics 630. We use TrustFL [5] as a comparison, which uses Merkle trees for commitment verification. In order to evaluate the performance of both methods, we consider three aspects of performance metrics, namely build time, query time and memory occupied. Build time refers to the time required to build the Merkel tree and dual Bloom filters after obtaining the hash summary. Query time refers to the time required to check whether the data exists in the set or not. Occupied Memory is the amount of memory required for the commitment in TEE. For FLoomChecker, the commitment is two Bloom filters. For TrustFL, the commitment is the root node of the Merkle tree.

We use MNIST dataset, which is a standard dataset widely used for handwritten digit recognition. The MNIST dataset contains 70,000 images, of which 60,000 images are used for training and 10,000 images for testing. Each image is a 28×28 pixel handwritten digit image, with digits ranging from 0 to 9, and the image is formatted with white characters on a black background. We set up 100 clients for training, and each client has 600 training samples. In each iteration, we select 10% of the clients to participate in the training.

B. Query Time

For the query time performance, we select Bloom filters and Merkel trees with false positive rates of 0.001 and 0.01, respectively. We use both LeNet-5 and MLP training models for comparison. In the experiments, the values of the local training rounds E are set to 10, 500, 1000, 3000, 5000, 8000, and 10000, respectively. In each experiment, we query whether each hash summary in these training rounds exists in the set, and divide the total query time by E to obtain the time required for a single query. To ensure the reliability of the experimental results, we conduct 100 experiments to take the average of the final results.

From the analysis of the complexity of the query time, the query time complexity of the Merkel tree is $O(\log E)$, where E is the number of rounds of local training. The query time complexity of the Bloom filter is $O(\xi)$, where ξ is the number of hash functions in the Bloom filter. The value of ξ is computed by the formula $\xi = -\ln p / \ln 2$, where p is the false positive rate. It can be seen that the query time of the Bloom filter depends on the magnitude of the false positive rate and is a constant level value that does not vary with the number of local training rounds. Whereas, the query time of Merkel

tree increases with the number of local training rounds.

The experimental results are shown in Fig. 3. It can be seen that the time required for one query of the FLoomChecker at two different false positive rates is always lower than that of the TrustFL, and the difference increases significantly with the increase of the value of E. The experimental results clearly show that the FLoomChecker significantly outperforms the TrustFL in terms of query time, especially with more local training rounds. With its constant-level query time complexity, the Bloom filter is able to provide more efficient verification performance while ensuring a lower false positive rate.

C. Build Time

In the build time performance comparison experiments, we chose the same data structure and model as the query time performance experiments. In the experiments, the values of local training rounds E are set to 10, 100, 200, 500, 1000, 1500, 2000, 3000, 4000, 5000, and 10000, respectively. During each round of training, we save the hash summaries of the model parameters for each round and build the data structure after all rounds are finished. To ensure the reliability of the experimental results, we perform the experiments of 100 times to take the average of the final results.

Table I shows the time taken by the two models of the model parameters to hash the model parameters under different training rounds. It can be seen that the hashing time of Lenet5 is consistently greater than that of MLP, due to the fact that Lenet5 has more parameters. However, it is important to note that the build time of these two approaches is independent of the model training time and the time spent on hashing the model parameters, as we built these two data structures after obtaining the summaries of the model parameter hashes for all rounds. The experimental results are illustrated in Fig. 4. It can be seen that the build time of FLoomChecker is always faster than that of TrustFL when the false positive rate is 0.1. With the increase of E value, the build time of TrustFL is gradually higher than that of FLoomChecker. We can obtain that under the same conditions, the FLoomChecker with a smaller false positive rate has a more obvious advantage in build time. This advantage is especially significant when there are more training rounds.

TABLE I: Time Spent on Hashing

Local training rounds	Lenet5 Time (ms)	MLP Time (ms)
10	0.7728	0.4064
100	6.7749	2.1897
200	12.9687	4.4007
500	32.9243	11.0710
1000	69.5618	22.9544
1500	103.0304	34.6060
2000	134.8485	44.8347
3000	208.2678	65.6738
4000	274.8791	85.3893
5000	344.0207	106.3500

D. Occupied Memory

In this experiment, we will measure the size of memory required for commitment by measuring the root node of a Merkel tree and a complete Bloom filter. We set the values of local training rounds E to 10, 100, 500, 1000, 3000, 5000, and 10000, respectively. The results of the experiment are shown in Fig. 5.

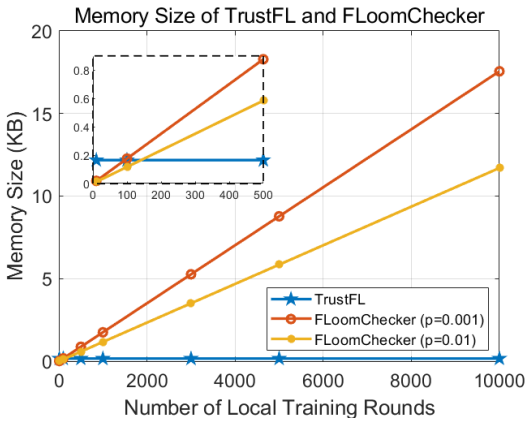

Fig. 5: Occupied Memory.

When the value of E is less than 100, FLoomChecker takes up less memory than the TrustFL. This is because at low rounds, the size of the Bloom filter is relatively small. However, the situation changes when E increases. Since the TrustFL only needs to pass the root node, the memory it occupies remains the same. In contrast, the memory usage of the Bloom filter gradually increases as E increases. Nevertheless, even when the number of training rounds reaches to 10000, the memory occupied by the Bloom filter with a false positive rate of 0.001 is only 17.55KB, which is a perfectly acceptable value for the TEE and does not significantly affect the performance of the TEE. It is worth mentioning that when the number of training rounds reaches to 10000, if the hash summaries of the model parameters for all rounds are stored directly into TEE, the memory required will reach to 945.38 KB. This data significantly proves the superiority of our approach in terms of memory utilisation.

E. Theoretical Value and Actual Value of the False Positive Rate

In order to verify the difference between the theoretical and actual values of the dual Bloom filters false positive rate, we generate a number of random strings and judge them after building the dual Bloom filters using hash digests with different rounds of E. The values of E are set to 10, 100, 200, 500, 1000, 2000, 3000, 5000, and 10,000, respectively. To ensure the reliability of the results, the experiments are repeated 100 times and the average value is taken as the final results. The experimental results are shown in Fig. 6. It can be seen that the false positive rate fluctuates within a small range above and below the theoretical value. This indicates that the difference between the theoretical and actual values is negligible.

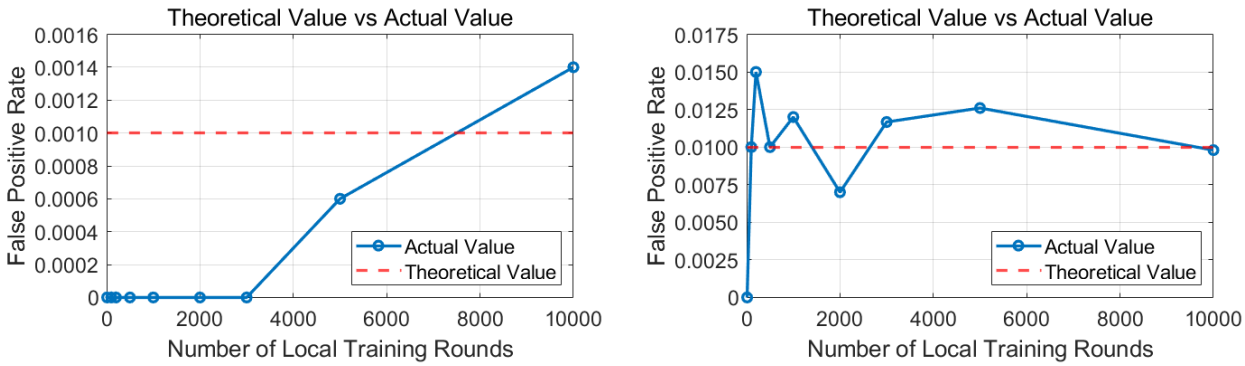

Fig. 6: Theoretical Value vs. Actual Value.

VII. Discussion

The Bloom filter is an efficient probabilistic data structure for determining whether an element belongs to a set or not. Despite its advantages of fast querying and space saving, its design still requires careful consideration of the false positive rate (i.e., the probability that a non-existing element may be misjudged as existing). As shown in Eq. (4) and Eq. (5), the size of the Bloom filter's bit array m and the number of hash functions ξ are both closely related to the false positive rate. A low false positive rate requires a larger bit array m and more hash functions ξ, which increases the storage and computation overheads. On the contrary, a high false positive rate reduces the storage and computation resources, but decreases the query accuracy.

VIII. Conclusion

In this paper, we have proposed a new approach called FLoomChecker that uses TEE and dual Bloom filters to verify the integrity of federated learning model training and ensure that all clients complete the training task truthfully. In order to ensure the freshness of the training data, we propose a dynamic input strategy. Through extensive experiments, we have demonstrated the correctness and effectiveness of the proposed FLoomChecker approach in verifying integrity of local model training.

Acknowledgment

This work was supported in part by the Natural Science Foundation of Shanghai (Grant No. 22ZR1400200), the Fundamental Research Funds for the Central Universities (No. 2232023Y-01), the National Natural Science Foundation of China (Grant No. 62472083, No. 62432008).

References

[1] M. Dai, T. Wang, Y. Li, Y. Wu, L. Qian, and Z. Su, "Digital twin envisioned secure air-ground integrated networks: A blockchain-based approach," *IEEE Internet of Things Magazine*, vol. 5, no. 1, pp. 96–103, 2022.

[2] J. Wang, X. Chang, R. J. Rodrìguez, and Y. Wang, "Assessing anonymous and selfish free-rider attacks in federated learning," in *2022 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–6, IEEE, 2022.

[3] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/Ispa*, vol. 1, pp. 57–64, IEEE, 2015.

[4] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[5] X. Zhang, F. Li, Z. Zhang, Q. Li, C. Wang, and J. Wu, "Enabling execution assurance of federated learning at untrusted participants," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 1877–1886, IEEE, 2020.

[6] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2018.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[8] W. Liu, H. Lin, X. Wang, J. Hu, G. Kaddoum, M. J. Piran, and A. Alamri, "D2mif: A malicious model detection mechanism for federated learning empowered artificial intelligence of things," *IEEE Internet of Things Journal*, 2021.

[9] A. Yazdinejad, A. Dehghantanha, H. Karimipour, G. Srivastava, and R. M. Parizi, "A robust privacy-preserving federated learning model against model poisoning attacks," *IEEE Transactions on Information Forensics and Security*, 2024.

[10] H. Jeong, H. Son, S. Lee, J. Hyun, and T.-M. Chung, "Fedcc: Robust federated learning against model poisoning attacks," *arXiv preprint arXiv:2212.01976*, 2022.

[11] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *International conference on machine learning*, pp. 5650–5659, Pmlr, 2018.

[12] J. Zhang, D. Wu, C. Liu, and B. Chen, "Defending poisoning attacks in federated learning via adversarial training method," in *Frontiers in Cyber Security: Third International Conference, FCS 2020, Tianjin, China, November 15–17, 2020, Proceedings 3*, pp. 83–94, Springer, 2020.

[13] H. Mozaffari, V. Shejwalkar, and A. Houmansadr, "Every vote counts:{Ranking-Based} training of federated learning to resist poisoning attacks," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1721–1738, 2023.

[14] Q. Xia, Z. Tao, Q. Li, and S. Chen, "Byzantine tolerant algorithms for federated learning," *IEEE Transactions on Network Science and Engineering*, vol. 10, no. 6, pp. 3172–3183, 2023.

[15] J. Lin, M. Du, and J. Liu, "Free-riders in federated learning: Attacks and defenses," *arXiv preprint arXiv:1911.12560*, 2019.

[16] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *arXiv preprint arXiv:1806.03287*, 2018.

[17] J. Kim, A. Biryukov, B. Preneel, and S. Hong, "On the security of hmac and nmac based on haval, md4, md5, sha-0 and sha-1," in *International Conference on Security and Cryptography for Networks*, pp. 242–256, Springer, 2006.