MUDGUARD
Taming Malicious Majorities in Federated Learning using Privacy-preserving Byzantine-robust Clustering

Wang, Rui; Wang, Xingkai; Chen, Huanhuan; Decouchant, Jérémie; Picek, Stjepan; Laoutaris, Nikolaos; Liang, Kaitai

DOI
10.1145/3700422

Publication date
2024

Document Version
Final published version

Published in
Proceedings of the ACM on Measurement and Analysis of Computing Systems

MUDGUARD: Taming Malicious Majorities in Federated Learning using Privacy-preserving Byzantine-robust Clustering

RUI WANG, Delft University of Technology, The Netherlands
XINGKAI WANG, Shanghai Jiao Tong University, China
HUANHUAN CHEN, Delft University of Technology, The Netherlands
JÉRÉMIE DECOUCHANT, Delft University of Technology, The Netherlands
STJEPAN PICEK, Radboud University, The Netherlands
NIKOLAOS LAOUTARIS, IMDEA Networks Institute, Spain
KAITAI LIANG, Delft University of Technology, The Netherlands

Byzantine-robust Federated Learning (FL) aims to counter malicious clients and train an accurate global model while maintaining an extremely low attack success rate. Most existing systems, however, are only robust when most of the clients are honest. FLTrust (NDSS '21) and Zeno++ (ICML '20) do not make such an honest majority assumption but can only be applied to scenarios where the server is provided with an auxiliary dataset used to filter malicious updates. FLAME (USENIX '22) and EIFFeL (CCS '22) maintain the semi-honest majority assumption to guarantee robustness and the confidentiality of updates. It is, therefore, currently impossible to ensure Byzantine robustness and confidentiality of updates without assuming a semi-honest majority. To tackle this problem, we propose a novel Byzantine-robust and privacy-preserving FL system, called MUDGUARD, to capture malicious minority and majority for server and client sides, respectively. Our experimental results demonstrate that the accuracy of MUDGUARD is practically close to the FL baseline using FedAvg without attacks (≈0.8% gap on average). Meanwhile, the attack success rate is around 0%-5% even under an adaptive attack tailored to MUDGUARD. We further optimize our design by using binary secret sharing and polynomial transformation, leading to communication overhead and runtime decreases of 67%-89.17% and 66.05%-68.75%, respectively.

CCS Concepts: • **Computing methodologies → Machine learning**; • **Security and privacy → Privacy-preserving protocols**; **Distributed systems security**.

Additional Key Words and Phrases: Federated Learning; Privacy Preservation; Robustness

ACM Reference Format:
Rui Wang, Xingkai Wang, Huanhuan Chen, Jérémie Decouchant, Stjepan Picek, Nikolaos Laoutaris, and Kaitai Liang. 2024. MUDGUARD: Taming Malicious Majorities in Federated Learning using Privacy-preserving Byzantine-robust Clustering. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 40 (December 2024), 41 pages. https://doi.org/10.1145/3700422

Authors' Contact Information: Rui Wang, r.wang-8@tudelft.nl, Delft University of Technology, The Netherlands; Xingkai Wang, starshine87@sjtu.edu.cn, Shanghai Jiao Tong University, China; Huanhuan Chen, h.chen-2@tudelft.nl, Delft University of Technology, The Netherlands; Jérémie Decouchant, J.Decouchant@tudelft.nl, Delft University of Technology, The Netherlands; Stjepan Picek, stjepan.picek@ru.nl, Radboud University, The Netherlands; Nikolaos Laoutaris, nikolaos.laoutaris@imdea.org, IMDEA Networks Institute, Spain; Kaitai Liang, kaitai.liang@tudelft.nl, Delft University of Technology, The Netherlands.

ACM 2476-1249/2024/12-ART40
https://doi.org/10.1145/3700422

Aggregation strategy	Threat model		Byzantine robustness	Updates confidentiality	No requirement for an auxiliary dataset	Computation complexity	Communication complexity
	Malicious server(s)	Malicious majority clients					
Zeno++ [63]	✗	✓	✓	✗	✗	$O(d)$	$O(nd)$
FLTrust [15]	✗	✓	✓	✗	✗	$O(n)$	$O(nd)$
FLAME [49]	✗	✗	✓	✓	✓	$O(d(S^2 + n^2))$	$O(d^2 + Sn^2)$
EIFFeL [55]	✗[1]	✗	✓	✓	✓	$O((n + d)n \log^2 n \log\log n + md \min(n, m^2))$	$O(n^2 + md \min(n, m^2))$
MUDGUARD (Ours)	✓	✓	✓	✓	✓	$O(d + n^3)$	$O(S(d + n^2))$

[1] EIFFeL considers a malicious server to be one that infers privacy information from other parties, which is equivalent to a semi-honest server in our context.

Table 1. Comparison of FL systems. d stands for the dimension of a model. n, m, and S represent the number of clients, malicious clients, and servers, respectively.

1 INTRODUCTION

Thanks to its privacy properties, Federated Learning (FL) [43] has been widely applied in real-world applications, e.g., prediction of the future oxygen requirements of symptomatic patients with COVID-19 [19]. Despite its attractive benefits, FL is vulnerable to Byzantine attacks. For example, attackers may choose to deteriorate the testing accuracy of models in an untargeted attack. Alternatively, they might fool models into predicting an attack-chosen label without downgrading the testing accuracy in a targeted attack. Many research works [23, 59, 62] have proved the vulnerability of FL via well-designed attack methods, e.g., poisoning training data or manipulating updates. Other studies [9, 15, 36, 44, 49, 55, 63, 64] have been dedicated to strengthening FL assuming that a minority of the clients can be malicious and that the server is honest. Beyond Byzantine attacks, FL could put clients at high risk of privacy breach [26, 65] even if clients' datasets are maintained locally. Several studies [49, 55, 57] have applied secure tools, e.g., Additive Homomorphic Encryption (AHE) [50], Differential Privacy (DP) [20, 60], and Secure Multiparty Computation (MPC), to protect clients' updates[1]. However, these works only guarantee security when all servers are (semi-)honest and when a minority of the clients are malicious.

To the best of our knowledge, there does not exist any FL system that is capable of withstanding the presence of a majority of Byzantine clients, as well as malicious servers, while also guaranteeing the confidentiality of updates. One may think that existing Byzantine-robust solutions could be trivially extended to address the above challenge. However, that is not the case because they either violate privacy preservation requirements or are only effective in the honest majority scenario. For example, FLTrust [15] and Zeno++ [63] require an auxiliary dataset that is independently and identically distributed (iid) with the clients' training datasets to rectify malicious updates, which evidently violates the clients' privacy. As for FLAME [49], it clusters updates and considers the smallest cluster as a malicious group, which makes sense in the malicious minority context. However, in the case of a malicious majority, it is difficult to assert if a given large/small-size cluster is malicious. EIFFeL [55] shows similar infeasibility, since it combines existing Byzantine-robust methods (e.g., FLTrust [15]) with secure aggregation [10].

Contributions. We propose a practical and secure Byzantine-robust FL system, MUDGUARD, that defends against malicious entities (i.e., malicious minority for servers and malicious majority for clients) with privacy preservation.

We summarize the advantages of MUDGUARD on the SOTA FL systems in Table 1. For a theoretical and empirical analysis of complexity, please refer to Appendix D and Section 5.2. Our main contributions can be described as follows.

[1]Note AHE and MPC require onerous computation over ciphertexts so that the computational complexity could naturally increase.

• We formulate a new aggregation strategy, *Model Segmentation*, for Byzantine-robust FL to effectively avoid poisoning attacks from a majority of malicious clients without requiring the servers to own an auxiliary dataset. It posits that the utilization of complex algorithms for the detection of malicious updates is not necessary. Instead, it only suggests implementing measures to prevent the co-existence of malicious and semi-honest clients in one aggregation.
• We propose a new method to improve the accuracy of updates clustering under non-iid scenarios. Instead of using the updates directly for clustering, we first compute the pairwise adjusted cosine similarity of updates (featured by different directions and magnitudes of updates between every two clients). Then, we input the results to DBSCAN.
• We design a secure FL system to be compatible with the cryptographic tools under the malicious context. To protect the updates on the server side and guarantee all clients receive correct aggregations, we construct a secure DBSCAN clustering that leverages cryptographic tools and secure aggregation with Homomorphic Hash Function (HHF) [24]. We further optimize the secure computations on the server side based on binary secret sharing and polynomial transformation.
• We provide a formal security proof for MUDGUARD in the UC framework. This proof captures dynamic security requirements, making MUDGUARD more practical than theoretical in security. MUDGUARD is the first UC-secure type in the research line of privacy-preserving FL.
• We implement MUDGUARD and perform evaluations on (F)MNIST and CIFAR-10 to quantify its accuracy under untargeted attacks, the Attack Success Rate (ASR) under targeted attacks or under an adaptive attack tailored to MUDGUARD, as well as its runtime and communication costs. Our experimental results show that the model trained by MUDGUARD maintains comparable testing accuracy with the FL baseline - a "no-attack-and-protection" FL with only honest parties (≈0.8% gap on average under untargeted attacks). The ASR under the targeted attacks is as low as 0%-5%. After optimizing the cryptographic computations, the runtime and communication costs are reduced by about 66.05%-68.75% and 67%-89.17%, respectively. For example, in the training of ResNet-18 using CIFAR-10, our optimization strategy can reduce training time from 95 seconds to 48 seconds and communication costs from 16331 MB to 5909 MB, whereas a vanilla FL takes nearly 24 seconds and 758 MB per round.

2 BACKGROUND AND RELATED WORK

2.1 Attacks against Federated Learning

Byzantine Attacks. Malicious clients may attempt to deteriorate the testing accuracy of the global model by intentionally uploading poisoned updates (i.e., untargeted attacks). Instead of harming the accuracy, the attackers may also intentionally use samples with triggers to launch attacks that make the model misclassify (i.e., targeted attacks). In the following, we review some classical and SOTA untargeted attacks (Gaussian Attack [23], Label Flipping Attack [8], Krum Attack and Trim Attack [23]) and targeted attacks (Backdoor Attack [3] and Edge-case Attack [59]).
• **Gaussian Attack (GA).** Malicious clients degrade the model accuracy by uploading local updates randomly sampled from a Gaussian distribution.
• **Label Flipping Attack (LFA).** Malicious clients flip the local data labels to generate faulty gradients. In particular, the label of each sample is flipped from y to $L - 1 - y, y \in [L]$, where L is the total number of classes.
• **Krum and Trim Attacks.** These two untargeted local model poisoning attacks are optimized for `Krum` [9] and `Trim-mean/Median` [64] aggregation strategies, respectively. They aim to pull the global model toward the opposite direction of the honest gradient when it is updated. Besides, they also have attack efficacy on `FedAvg`.
• **Backdoor Attack (BA).** Byzantine clients embed triggers to training samples and change their

labels to targeted labels. Their goal is to make the global model misclassify the correct labels to the targeted ones when testing samples with triggers.

• **Edge-case Attack (EA).** The attack aims to misclassify seemingly similar inputs that are unlikely to be part of the training or testing data. For example, by labeling Ardis[2] "7" images as "1" and adding them to training data, EA can easily backdoor an MNIST classifier. Similarly, the attack can use a Southwest airplanes dataset labeled "truck" to inject a backdoor into a CIFAR-10 classifier. Note that the attack relies on a restricted assumption that an extra dataset resemblance to the training dataset should be given.

Inference Attacks. Although local datasets are not directly revealed during the FL training process, the updates are still subject to privacy leakage if the server is semi-honest or even malicious [42, 47, 65]. For instance, Zhu *et al.* [65] investigated a method of training data reconstruction via optimizing the L_2 distance between uploaded gradients and gradients trained from dummy samples using an L-BFGS solver. This approach allows servers to easily reconstruct the local datasets and achieves even pixel-wise accuracy for images and token-wise matching accuracy for texts.

Differential Attack. We use DBSCAN in conjunction with *Model Segmentation* to separate benign and malicious updates. Features extraction with adjusted cosine similarity greatly decreases the likelihood of false positives. However, we cannot guarantee that the clustering results are 100% correct. A benign update may be erroneously clustered with malicious updates due to a marginal probability, resulting from its potentially lower similarity to benign updates compared to malicious updates. In particular, this case happens more frequently in SignSGD, where only taking signs of gradients to update the model because the algorithm computing adjusted cosine similarity disregards the magnitude of the gradients, resulting in the same effect as calculating cosine similarity. The above phenomenon triggers the differential attack in the following cases. Assuming that, at t-th round, if a benign update is incorrectly clustered with malicious updates during aggregation, the server combines them. Subsequently, by subtracting the malicious updates from the returned aggregation, the benign update becomes easily revealed, facilitating the launch of an inference attack. Another more common case is that a malicious adversary $\mathcal{A}$ compromises m clients and then makes one of them perform correct operations, i.e., acting benignly. This malicious-but-act-benign client, being assigned to a benign group, can get benign aggregation from each round and then conduct an inference attack.

2.2 Defenses against Federated Learning

Byzantine-robust Federated Learning. Blanchard *et al.* [9] proposed `Krum` to select 1 out of n (local updates) as a global update for each round, where the selected updates should have the smallest L_2 distance from others. Yin *et al.* [64] introduced `Trim-mean` and `Median` to resist Byzantine attacks. The former uses a coordinate-wise aggregation strategy. The server calculates $n - 2z$ values for each model parameter as the global update, wherein the largest and smallest z values are filtered. Unlike `FedAvg` [43] computing the weighted average of all parameters, the latter calculates the median of parameters. This median serves as an update to the global model. A major drawback of the aforementioned mechanism is that it is effective only under a majority of honest clients working with an honest server. In `Median` [64], the median calculated by the server can easily be malicious if malicious clients control a large proportion of updates. This similarly applies to `Trim-mean` and `Krum`. Cao *et al.* [15] proposed `FLTrust` to protect against a malicious majority at the client side, assuming an honest server holds a small auxiliary dataset. The server treats the gradients trained from this small dataset as the root of trust. By comparing these trusted results

[2]A dataset extracted from 15,000 Swedish church records written by different priests with various handwriting styles in the nineteenth and twentieth centuries.

with the updates sent by clients, the server can easily rule out malicious updates. Under the same assumption, Zeno++ [63] uses an auxiliary dataset to calculate the loss value of each local model. A client is determined to be honest if the loss value is beyond the preset threshold. While using an auxiliary dataset could be intriguing, such approaches are not feasible in the context of FL as they violate the fundamental premise of FL in which local datasets are not to be shared with any parties.
Privacy-preserving Federated Learning. Truex *et al.* [57] proposed a solution enabling clients to use AHE and DP to secure gradients in the semi-honest context (for both clients and the server). Since DP noise is applied on gradients, the accuracy of the global model is deteriorated. In the scenario of honest majority clients with two semi-honest servers, Thien *et al.* [49] proposed FLAME using an MPC protocol to protect gradients from the servers and enabling the servers to perform clustering for Byzantine robustness. Specifically, the clients can securely share their updates to the servers cryptographically, e.g., via secret sharing, and the servers can filter out malicious updates without knowing their concrete values. By expressing existing Byzantine-robust solutions (e.g., FLTrust) as arithmetic circuits, EIFFeL [55] enables secure aggregation of verified updates. Although FLAME and EIFFeL capture both Byzantine robustness and privacy preservation (i.e., update confidentiality), the accuracy of the global model could become equivalent to a random guess if the proportion of malicious clients is $\geq 50\%$.
Cryptographic Tools. Secret Sharing (SS) is a method for splitting a secret among multiple clients such that no individual can reconstruct the secret without a subset of the shares. Shamir Secret Sharing (SSS) is one such scheme, with linear properties enabling efficient computations in FL that allow t out of n to reconstruct the secret. Homomorphic Encryption (HE) is a privacy-preserving technique that allows computations on ciphertexts without exposing plaintexts. It supports either partial (addition or multiplication) or full (both operations) homomorphic properties. Homomorphic Hash Functions (HHF) use a collision-resistant hash function supporting additive homomorphism, used for verifying the correctness of aggregation. Oblivious Transfer (OT) ensures that a receiver can learn only one of several sender-held strings without the sender learning which string is selected. Garbled Circuits (GC) enable two servers to securely evaluate a Boolean circuit, where the garbler generates a garbled version of the circuit, and the evaluator computes the output without revealing intermediate values. Due to the page limit, we review machine learning and the detailed definitions of security tools in Appendix C.

3 PROBLEM FORMULATION

3.1 System Model

Before proceeding, we provide some assumptions about MUDGUARD. We assume training is conducted on a dataset $\mathcal{D}$ with K data samples composed with feature space $\mathcal{X}$ (each sample containing all features) and a label set $\mathcal{Y}$. Additionally, $\mathcal{D}$ is horizontally partitioned among n clients, indicated as $\mathcal{X}_i = \mathcal{X}_j, \mathcal{Y}_i = \mathcal{Y}_j, \mathcal{I}_i \cap \mathcal{I}_j = \emptyset, \forall \mathcal{D}_i, \mathcal{D}_j, i \neq j$, where all clients share the same feature space and labels but differ in sample index space $\mathcal{I}$. FL aims to optimize a loss function: $\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \frac{k_i}{K} \mathcal{L}_i(\boldsymbol{w}, \mathcal{D}_i)$, where $\mathcal{L}_i(\cdot)$ and k_i are the loss function and local data size of i-th client.

For reasons that relate to the versatility of the FL system, we also consider S (> 2) servers to carry out clustering and aggregation (e.g., FedAvg). This allows us to protect from malicious servers who cannot reconstruct the secrets so long as their number is less than $S/2$ by using cryptographic tools, in which clients send updates in secret-shared format. We state that our Byzantine solution can also be executed by only one server. In this case, considering privacy, we have to assume that the server must be fully trusted or semi-honest. Note that our focus here is on the existence of malicious servers. In this research line [39, 46], secure computation is considered among multiple servers. Due to page limit, we summarize frequently used notations in Table 6 (see Appendix A).

3.2 Threat Model

Attackers' goal. We assume that two different entities are involved in the training: semi-honest and dynamic malicious parties (including servers and clients), in which both try to infer the privacy (updates) information of others from the received messages. Unlike the former, which strictly follows the designed algorithms, the malicious clients additionally aim to deteriorate the performance or boost the ASR of the global model through untargeted or targeted poisoning attacks, respectively.
Attackers' capabilities. The malicious servers (in a minority proportion) and clients (in a majority proportion) can deviate from the designed protocols. For example, the malicious servers can perform an incorrect aggregation and send it back to the semi-honest group. Moreover, malicious parties (servers and clients) can collude with each other to infer benign aggregations and maximize the efficacy of poisoning attacks (e.g., the Krum attack). To resist outside adversaries, secret-shared messages are transmitted by private communication channels. Other messages are transmitted through public communication channels, where outsiders are allowed to eavesdrop on these channels and try to infer clients' (updates) privacy during the whole training phase.
Attackers' knowledge. We assume that the loss function, data distributions, Byzantine-robust aggregation strategy, and public parameters (including training and security parameters) are revealed to all parties. The malicious clients can exploit this information to design and cast adaptive attacks tailored to MUDGUARD. For privacy reasons, the local updates and datasets of semi-honest clients are not revealed to malicious parties.

4 MUDGUARD OVERVIEW AND DESIGN

4.1 Overview

The first goal of this work is to maintain the Byzantine robustness such that malicious updates should be excluded properly. To do so, the servers must separate the malicious clients from the semi-honest clients. DBSCAN helps the servers to perform clustering. Since the main difference between the malicious and the benign is in the direction and magnitude of the updates, we use the adjusted cosine similarity of updates as feature extraction to obtain better clustering accuracy. Under the (semi-)honest majority, the clustering result directly links to the group size. However, for a dynamic malicious majority, we cannot judge if a cluster is malicious only based on its size. To address this issue, we propose *Model Segmentation*. Unlike traditional FL generating "a unique" global model, our proposed algorithm can yield multiple aggregation results. It does not require the servers to know whether a given group is malicious or not. Moreover, it only aggregates the updates within the same cluster and then returns the results to the corresponding clients. We thus guarantee that the semi-honest will not be aggregated with the malicious.

As far as fighting against inference attacks is concerned, we should protect the confidentiality of the updates. For this, we use SS to wrap the updates into a secret shared format in the sense that individual secret shares cannot reveal the underlying information of the updates. By doing so, we guarantee that the updates are secured from eavesdroppers, semi-honest, or even malicious servers and can further be used on secure multiplication, comparison, and aggregation via cryptographic tools. However, using SS alone is not sufficient to defend against differential attacks. To thwart the attack, we apply DP to prevent the attackers from extracting benign updates from the semi-honest group. Since injecting noise has a negative influence on the accuracy of the training model, we enable clients to perform denoising before wrapping the results into shares. Note that this does

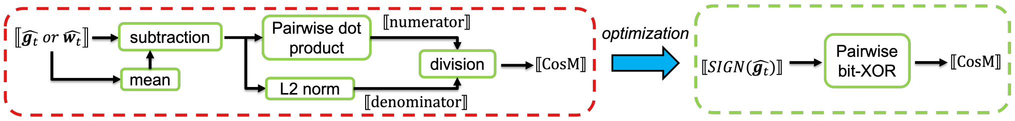


Fig. 1. Optimization on the calculation of adjusted cosine similarity. - - - - : optimized process: by binary SS, [[CosM]] is computed via bit-XOR. - - - - : unoptimized process: taking gradients or weights as updates to compute [[CosM]].

not invalidate DP due to the post-processing nature [20]. We also consider the malicious minority servers and thus leverage HHF to prevent malicious servers from performing incorrect aggregation, e.g., merging the gradients from two different groups.

4.2 Byzantine-robust Aggregation Strategy with Cryptographic Computations

Our workflow for the Byzantine-robust aggregation strategy is as follows. Firstly, the clients upload the gradients of the local models to the server side. Secondly, servers extract features of gradients and split gradients into multiple clusters via DBSCAN. Finally, servers aggregate the gradients in the clusters separately and send aggregations to the corresponding clients. In the following, we complete the strategy over secure cryptographic computations.

Gradients Upload. Consider computing pairwise adjusted cosine similarity matrix (CosM) as the inputs of DBSCAN. Client i first subtracts updates with their mean values $g_{i,0} - \bar{g}_i$, where $o \in O$ is the index of updates and uploads them to the servers. Servers compute the pairwise dot product and L_2 norm to derive the numerator and denominator, respectively. Then for any two clients i, j, servers can calculate $\mathrm{CosM} := \{\mathrm{CosM}_{ij} = \frac{\sum_{o\in O}(g_{i,o}-\bar{g}_i)(g_{j,o}-\bar{g}_j)}{\sqrt{\sum_{o\in O}(g_{i,o}-\bar{g}_i)^2}\cdot\sqrt{\sum_{o\in O}(g_{j,o}-\bar{g}_j)^2}}\}$ from the division. The use of the CosM as a method for extracting features is motivated by the fact that it measures both the difference in directions and magnitudes of updates. This is particularly useful when dealing with clients exhibiting various behaviors and non-iid cases. In this context, CosM and L_2 distance are used as input and the metric of DBSCAN, respectively.

Beyond Straightforward Cryptographic Combinations. Recall that one of our goals is to preserve the privacy of clients. Trivially applying cryptographic building blocks to realize the goal will easily yield expensive computation and communication complexity, as the costs performed over ciphertext inherently demand a substantial amount of resources compared to plaintext counterparts. Furthermore, the arithmetic operations involve substantial (non-)linear computations. The servers should perform the computations of shared mean, numerator, denomination, and then division to finally get the shared adjusted cosine similarity matrix [[CosM]]. To optimize the efficiency of the above, we consider the denoised gradients of client i at t-th round $\hat{\mathbf{g}}_t^i$ as updates and perform binary secret sharing via SignSGD. Note that SignSGD only takes the signs of gradients to the update model, resulting in benign and malicious having the same magnitudes. Thus, in this case, we can easily compute the adjusted cosine similarity via simple bit-wise XORing. Figure 1 depicts this optimization procedure. The empirical comparison will be presented in Section 5.2.

Therefore, in each training round (of the optimization), client i derives the gradients using SGD [11]. Considering the upcoming cryptographic clustering, one needs to compute the signs of gradients $\mathrm{sign}(\hat{\mathbf{g}}_t^i) \in \{-1, +1\}$ as SignSGD [6] and then encodes to Boolean representation, which is compatible with binary SS and XOR operations. Without loss of generality, we implement a widely used encoding/decoding method as

$$\mathsf{ECD}(\mathsf{sign}(\hat{\mathbf{g}}_t^i)) = \begin{cases} 1, & \mathsf{sign}(\hat{\mathbf{g}}_t^i) = +1 \\ 0, & otherwise \end{cases}, \mathsf{DCD}(\mathsf{ECD}(\mathsf{sign}(\hat{\mathbf{g}}_t^i))) = 2\mathsf{ECD}(\mathsf{sign}(\hat{\mathbf{g}}_t^i)) - 1.$$

This method guarantees $\mathsf{DCD}(\mathsf{ECD}(\mathsf{sign}(\hat{\mathbf{g}}_t^i))) = \mathsf{sign}(\hat{\mathbf{g}}_t^i)$. Each client sends the encoded updates to the servers via binary SS and broadcasts the hash results of unencoded updates for future verification. Although SignSGD is lightweight, it has a negative impact on the accuracy of clustering. Section 5.1 provides a detailed analysis of this impact. Note that SignSGD has the natural capability of defending against scaling attacks [3] since it only takes signs as updates and clips the magnitude of gradients. An attacker can still easily deteriorate the global model by constructing updates in the opposite direction of benign updates.

Clustering. As a crucial variable in FL, updates determine the directions and magnitudes of updating in the model, while Byzantine attackers introduce abnormal updates. Traditional clustering approaches directly use updates as inputs and cosine similarity as metric [49], causing informative redundancy and blurring obvious features, especially in deep models (e.g., ResNet [30]), thereby producing frequent false positives and negatives. Since the adjusted cosine similarity measures the difference in directions and magnitudes of updates at the same time, we use the pairwise adjusted cosine similarity CosM as a method for extracting features, i.e., CosM and L_2 distance, used as the input and the main metric of DBSCAN, respectively. We find that this method is effective in distinguishing the updates because 1) calculating CosM (feature extraction) is equivalent to reducing the informative redundancy of updates to improve the clustering accuracy. 2) Using it to calculate the pairwise L_2 distance can expand the pairwise differences that are not clearly computed by CosM. Thus, there is a more clear density difference between honest and malicious updates. 3) By subtracting the mean updates, CosM helps to account for reducing the influence of non-iid, allowing for a more accurate comparison of clients' updates. 4) When the model converges, using cosine similarity is inappropriate because even semi-honest clients have updates in different directions. For example, under a Gaussian Attack (GA), the malicious and semi-honest clients become indistinguishable. The accuracy of the global model drops sharply to the level of the initial training. We provide concrete examples in Appendix I.2 to demonstrate the advantages of using CosM. Note that this advantage is more notable when the cryptographic tools are not optimized. Since the proposed optimization uses SignSGD to align the magnitudes of updates, computing cosine similarity on it naturally provides the same effect on clustering as adjusted cosine similarity.

Next, we describe the process of our clustering. We first extract features (different updates directions with magnitudes) - calculating the $\mathsf{CosM} \leftarrow \overline{\mathbf{g}}_t^i \oplus \overline{\mathbf{g}}_t^j, i, j \in [n]$, and then use it as the input for clustering, thereby reducing the rate of false positives. Commonly, the adjusted cosine similarity of two vectors is obtained by first calculating the dot product of the vectors and then dividing them by the product of their respective L_2 norm. Since encoding updates $\hat{\mathbf{g}}_t^i \in \{-1, +1\}^d$ to $\overline{\mathbf{g}}_t^i \in \{0, +1\}^d$ is inspired by [53], we compute XOR of $\overline{\mathbf{g}}_t^i$ with $d - 2p$ bits, which is equivalent to the result of the dot product of $\hat{\mathbf{g}}_t^i$, where p is the counted number of set bits.

After that, the servers collaboratively compute pairwise L_2 distance matrix EucM using secure multiplications ($vector_{ij} \leftarrow \mathsf{CosM}_i - \mathsf{CosM}_j, i, j \in [n]$, $x_{ij} \leftarrow vector_{ij} \cdot vector_{ij}$, and $\mathsf{EucM}_{ij} \leftarrow 1 + \frac{x_{ij}-1}{2} - \frac{(x_{ij}-1)^2}{8} + \frac{(x_{ij}-1)^3}{16}$) and compare with density α to derive an indication matrix IndM, where $\mathsf{IndM} = 1$ if $\mathsf{EucM} \leq \alpha$, otherwise $\mathsf{IndM} = 0$. Then, by applying the DBSCAN, one can derive cluster labels. Note that the main focus of this paper is not on optimizing DBSCAN, we thus do not describe how to retrieve cluster labels from IndM. We refer interested readers to [22].

We see that α has a crucial influence on clustering accuracy. Huang *et al.* [32] concluded that a well-trained model follows an alike-Gaussian distribution. We use this finding to guide the selection of this hyperparameter and formally derive its upper bound of selection (see Theorem 4.1 and its proof in Appendix G).

Theorem 4.1 (Density Selection). *Suppose the distribution of benign and malicious updates obeys the normal distribution; setting $\alpha < \sqrt{2}$ guarantees that malicious clients conducting a poisoning attack will not be grouped together with benign clients.*

Note that taking $\alpha < \sqrt{2}$ only allows malicious clients to be identified as noise points, which is not a 100% guarantee that all semi-honest clients are clustered together. Due to the difference in training data, it could happen that the distances between a semi-honest client and other semi-honest clients are greater than $\sqrt{2}$ by chance, resulting in the semi-honest client being identified as a noise point.

Model Segmentation. To deal with Byzantine-majority attacks, after obtaining the cluster labels, the servers aggregate the updates within the same cluster and return the results (and their hash values) to the corresponding clients. In our design, unless a malicious client acts honestly, then it will not be grouped into a cluster with the semi-honest clients, with a relatively large probability. This protects benign clients by keeping poisonous updates from global model updates computed for benign clusters. Note here we do not further explore the case in which malicious clients choose to act honestly during training. In fact, if malicious clients behave semi-honestly, we will obtain a more accurate global model. In a sense, this is a bonus for semi-honest clients. After all, in the context of *Model Segmentation*, it is not required to identify malicious groups via any verification algorithms, which is a positive thing since it removes the processing burden from the servers as the latter do not need to run verification over “encrypted-and-noised" updates. Compared with the method of FLAME, *Model Segmentation* does not need to assume that most clients in the FL system are (semi-)honest. When integrated with an optimized clustering method, *Model Segmentation* can also enhance the Byzantine robustness of the MUDGUARD.

Resistance against Malicious Servers. To further prevent malicious servers from casting and sending incorrect aggregation, we use HHF so that every client can verify if the received aggregation is correct. Specifically, the proposed method involves a pre-upload step in which client i broadcasts hash values of signs of gradients $\mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\boldsymbol{g}}t^i))$ to the remaining parties before uploading secret-shared updates to the server side. The servers use additive homomorphism of HHF to calculate hash values of aggregations $\prod_{i \in c_j} \mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\boldsymbol{g}}t^i)) = \mathsf{H}_{\delta,\phi}(\boldsymbol{G}_t^j)$ based on the clustering results, where c_j refers to a cluster j containing client indexes. After receiving the aggregations $\boldsymbol{G}_t^j$, the clients can calculate $\mathsf{H}_{\delta,\phi}(\boldsymbol{G}_t^j)$ and $\prod_{i \in c_j} \mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\boldsymbol{g}}_t^i))$ based on the cluster labels and the received hash values, and subsequently verify whether these two values are equal or not. Note that this method considers the possibility of malicious servers that may send incorrect aggregations and IndM to the semi-honest clients. However, since only a minority of servers are assumed to be malicious, the semi-honest clients take the most consistent results as the real results.

Comparison with FLAME. Intuitively, systems that use clustering, like FLAME [49], could experience misclassification problems under non-iid cases. Different from FLAME using the updates directly for clustering, we first compute the pairwise adjusted cosine similarity of updates. Then, we push the results to DBSCAN. This is a crucial step in feature extraction, improving the accuracy of clustering (see Table 4). We further optimize communication and computation complexity. In our design, MPC impacts three main stages: computations of CosM, L_2 distance, and element-wise comparison. Since the 1*st* stage (i.e., computations of CosM) is the most computationally expensive, we focus on its optimization (see above). After the optimization, the servers avoid using “heavy" tools, like HE and Beaver’s multiplication, to calculate CosM so that communication and computational overheads are reduced. Specifically, MUDGUARD calculates CosM by doing XOR locally on servers, and the matrix size reduces from the number of clients × the number of model updates ($n \times d$) to $n \times n$ after calculation. Then, this $n \times n$ matrix is used to calculate the pairwise-L_2 distance (where only multiplication is involved). While in FLAME, directly calculating cosine similarity requires multiplication, division, and the square root of the $n \times d$ matrix, which are relatively intensive, expensive operations in MPC.

4.3 System Design

Assume client $i \in [n]$ holds a horizontally partitioned dataset $\mathcal{D}_i$ satisfying $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i$, at t-th round, MUDGUARD works as follows.

Protocol MUDGUARD

❶ *Local Training.* For each local minibatch, each client conducts SGD and takes gradients $\mathbf{g}_t^i$ as updates.
❷ *Noise Injection.* Each client adds noise into $\mathbf{g}_t^i$ to satisfy DP: $\widetilde{\mathbf{g}}_t^i \leftarrow \mathbf{g}_t^i / \max(1, \|\mathbf{g}_t^i\|_2/\Delta) + \mathcal{N}(0, \Delta^2\sigma^2)$.
❸ *Denoising.* To improve accuracy, each client denoises $\widetilde{\mathbf{g}}_t^i$ by $\hat{\mathbf{g}}_t^i \leftarrow \mathsf{KS}(\widetilde{\mathbf{g}}_t^i, \mathcal{N}) \cdot \widetilde{\mathbf{g}}_t^i$, where $\mathsf{KS}(\cdot)$ is the KS distance.
❹ *SS.* Each client splits $\overline{\mathbf{g}}_t^i \leftarrow \mathsf{ECD}(\mathsf{sign}(\hat{\mathbf{g}}_t^i))$ into S shares by binary SS with Tiny Oblivious Transfer (OT) and sends the shares to S servers: $[[\overline{\mathbf{g}}_t^i]] \overset{SS}{\longleftarrow} \overline{\mathbf{g}}_t^i$. Besides, by running HHF, all clients broadcast $\mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\mathbf{g}}_t^i))$.
❺ *Feature Extraction.* After receiving n shares, each server locally computes a pairwise adjusted cosine similarity matrix by bit-XOR: $[[\mathsf{CosM}_{ij}]] \leftarrow [[\overline{\mathbf{g}}_t^i]] \oplus [[\overline{\mathbf{g}}_t^j]], i, j \in [n]$. To further compute L_2 distance, all servers convert Boolean shares to arithmetic shares by correlated randomness.
❻ *L_2 Distance Computation.* After conversion, deriving multiplicative SS, each server uses HE or OT to produce a triple, satisfying further multiplications. Therefore, each server takes $[[\mathsf{CosM}]]$ as the inputs of DBSCAN and then computes $[[\mathsf{EucM}]]$ by (a) pairwise subtraction: $[[vector_{ij}]] \leftarrow [[\mathsf{CosM}_i]] - [[\mathsf{CosM}_j]], i, j \in [n]$, (b) dot product: $[[x_{ij}]] \leftarrow [[vector_{ij}]] \cdot [[vector_{ij}]]$, and (c) approximated square root: $[[\mathsf{EucM}_{ij}]] \leftarrow 1 + \frac{[[x_{ij}]]-1}{2} - \frac{([[x_{ij}]]-1)^2}{8} + \frac{([[x_{ij}]]-1)^3}{16}$.
❼ *Element-wise Comparison.* By comparing each element of EucM with density parameter α, each server can derive shares of indicator matrix $[[\mathsf{IndM}]]$, $\{\mathsf{IndM}_{ij} = 1 \mid \mathsf{EucM}_{ij} \leq \alpha\}$.
❽ *Reconstruction.* All servers run a reconstruction algorithm to reveal IndM: $\mathsf{IndM} \overset{\text{recon}}{\longleftarrow} [[\mathsf{IndM}]]$ and broadcast it to the client side. By DBSCAN, one can derive cluster labels. Based on these labels, the clients learn about clustering information to perform aggregation verification in step ❿.
❾ *Model Segmentation.* The servers aggregate shares (based on the number of labels c) with the same labels after decoding: $\{[[\boldsymbol{G}_t^j]] \leftarrow \sum_{i \in c_j} \mathsf{DCD}([[\overline{\mathbf{g}}_t^i]]) \mid c_j = \{i \mid i \in [n]\}, j \in [c],\}$ and send to the corresponding clients.
❿ *Aggregation Verification.* After reconstructing aggregation, according to cluster labels, each client verifies aggregation by $\prod_{i \in c_j} \mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\mathbf{g}}_t^i)) \overset{?}{=} \mathsf{H}_{\delta,\phi}(\boldsymbol{G}_t^j)$. If the equation holds, clients accept the aggregation results; otherwise, reject and abort.

We note that the corresponding implementation-level algorithms of MUDGUARD are given in Appendix B and will be used in the experiments.

4.4 Privacy Preservation Guarantee

Differential attack resistance. As shown in step ❷ and ❸ of Protocol 4.3, each client i can add differentially private noise into gradients and perform denoising later. Like [48], we use KS distance (of noised gradients and noise distribution) as a metric to denoise by multiplying noised gradients. Differentially private updates are first denoised, taken signs, and encoded before being secretly shared.

Binary SS. Unlike arithmetic SS in domain $\mathbb{Z}_{2^b}$, binary SS works with $b = 1$, where b is the bit length. To resist malicious clients deviating from SS specifications, we apply OT in our design (step ❹ of Protocol 4.3). However, this brings a considerable increase in communication costs. Furukawa *et al.* [25] used TinyOT to generalize multi-party shares with communication complexity linear in the security parameter. We follow this method so that each client i binary shares its updates to S servers. The SS scheme guarantees that a malicious server cannot reconstruct the secret even if colluding with the rest of the servers under a malicious minority setting.

XOR. In step ❺ of Protocol 4.3, after receiving shares, each server can compute the pairwise dot product independently. Assume a server s has $[[\overline{\mathbf{g}}_t^i]]_s$, where $s \in [S]$. Since $\overline{\mathbf{g}}_t^i = [[\overline{\mathbf{g}}_t^i]]_1 \oplus \cdots \oplus [[\overline{\mathbf{g}}_t^i]]_S$, we have $\overline{\mathbf{g}}_t^i \oplus \overline{\mathbf{g}}_t^j = [[\overline{\mathbf{g}}_t^i]]_1 \oplus [[\overline{\mathbf{g}}_t^j]]_1 \cdots \oplus [[\overline{\mathbf{g}}_t^i]]_S \oplus [[\overline{\mathbf{g}}_t^j]]_S, \forall i, j \in n$. Therefore, in this case, each server s can compute $[[dot\_product]]$ by $\{[[dot\_product_{ij}]]_s = [[\overline{\mathbf{g}}_t^i]]_s \oplus [[\overline{\mathbf{g}}_t^j]]_s \mid \forall i, j \in [n]\}$ locally and without interactions with other servers. By multiplying a constant, one can derive shares of adjusted cosine similarity. Using binary SS can help us to save element multiplication and division operations.

Bit to Arithmetic Conversion. The servers also need to convert the shares in $\mathbb{Z}_2$ to arithmetic shares ($\mathbb{Z}_{2^b}$) to support the subsequent linear operations and multiplications. We implement the conversion by following [54]. A common method is to use correlated randomness in these two domains (doubly-authenticated bits) and extend them. After this, the servers can derive arithmetic shares of the dot product. Note some works [2, 45] leverage straightforward transformation under the cases with only semi-honest parties.

Multiplication. As shown in step ❻ of Protocol 4.3, multiplications are necessary in DBSCAN. Considering the semi-honest majority setting on the server side, the replicated SS and SSS can be applied here since both satisfy the multiplicative property, in which two share multiplications can be computed locally without any interaction. For the existence of malicious servers, we consider the protocol proposed by Lindell *et al.* [40], modifying SPDZ [18] to the setting of multiplicative secret sharing modulo a prime (including replicated SS and SSS). Furukawa *et al.* [25] also proposed a similar variant for TinyOT. Both are based on the observation that the optimistic triple production using HE or OT can be replaced by producing a triple using multiplicative secret sharing instead.

Secure Comparison with Density α. With arithmetic shares, the comparison (step ❼ of Protocol 4.3) requires extra correlated randomness, especially secret random bits in the larger domains. For the semi-honest majority servers, we follow the protocol [16] with $\mathbb{Z}_2$ to implement comparison efficiently. Under the malicious minority, we should check if the output is actually a bit. We follow [17] to multiply a secret random bit with comparison output and then reconstruct it. If the reconstructed value is a bit, it proves that the malicious servers do not deviate from the comparison protocol.

4.5 Security Analysis

MUDGUARD achieves security properties under *malicious majority* clients and *malicious minority* servers. Malicious parties may arbitrarily deviate from the protocol, while the rest of the parties are semi-honest, trying to infer information as much as possible (but following the protocol). We assume malicious clients and servers may collude with each other.

A secure FL system satisfies correctness, privacy, and soundness. The latter two are security requirements. Informally, the requirements are: (1) the adversary learns nothing but the differentially private output; (2) the adversary cannot provide an invalid result accepted by a benign client. We first define the security in the UC framework [14]. This allows the system to remain secure and capable of being arbitrarily combined with other UC secure instances. In such a framework, security is defined by a well-designed ideal functionality that captures several properties simultaneously, including correctness, privacy, and soundness. Specifically, Figure E (Appendix E) shows our ideal functionality $\mathcal{F}_{\text{MUDGUARD}}$. The definition captures all required security properties except DP and soundness against malicious clients. Appendix E will discuss the remaining.

We analyze the security in an $\mathcal{F}$-hybrid model, where $\mathcal{F}$ denotes a set of ideal functionalities that the protocol can access. Our protocol adopts three ideal functionalities: $\mathcal{F}_{\text{RO}}$, $\mathcal{F}_{\text{SS}}$ and $\mathcal{F}_{\text{B2A}}$. The $\mathcal{F}_{\text{RO}}$ is an ideal functionality that models a random oracle. The latter two, $\mathcal{F}$SS and $\mathcal{F}$B2A, are ideal functionalities representing a secure SSS [25] and a bit-to-arithmetic conversion [54], respectively. These functionalities model the secure handling of shared secrets, bit-to-arithmetic operations, and random oracle queries. Furthermore, the works in [25] and [54] provide concrete constructions that can securely realize $\mathcal{F}_{\text{SS}}$ and $\mathcal{F}_{\text{B2A}}$, demonstrating the practical feasibility of our proposed model. With these ideal functionalities in place, we present the following theorem to formally state the UC security of MUDGUARD:

Theorem 4.2. *Assuming the existence of a secret sharing scheme that can UC-realize $\mathcal{F}_{SS}$ and a bit-to-arithmetic conversion scheme that can UC-realize $\mathcal{F}_{B2A}$, the protocol* MUDGUARD *is UC-secure in the random oracle model. Specifically,* MUDGUARD *securely realizes $\mathcal{F}_{\text{MUDGUARD}}$ in the ($\mathcal{F}_{RO}$, $\mathcal{F}_{SS}$, $\mathcal{F}_{B2A}$)-hybrid model, in the presence of a malicious-majority of clients and a malicious-minority of servers, allowing for arbitrary collusions between malicious parties.*

The remaining two properties are related to data output, which is not concerned with the cryptographic view. Specifically, DP is provided by adding noise (Appendix E), and soundness against malicious clients is provided by *Model Segmentation.*

Note our well-designed functionality captures as many attacks as possible. In other words, soundness against malicious clients and DP cannot be achieved under the UC model. On the one hand, recognizing malicious clients is quite a *subjective* task since they do not deviate from the protocol in cryptographic ways. There might be a benign client providing similar inputs that seem to be malicious, with a non-negligible possibility. On the other hand, the output with DP can be obtained by the adversary in our definition. Hence, differential attacks should not be captured in the functionality.

4.6 Adaptive attack

Recall that in Section 3.2, a Byzantine-robust aggregation strategy is available to attackers. Malicious clients can adapt their attacks to nullify the robustness of the system. Note that untargeted attacks (e.g., Krum and Trim attacks) solve an optimization problem to maximize the efficacy of attacks, meaning the strategies of untargeted attacks are already optimal. Therefore, we design and evaluate an adaptive backdoor attack for MUDGUARD. Specifically, the attack is formulated by adding a sub-task to the attack optimization problem. Given the fact that MUDGUARD achieves Byzantine-robustness by aggregating only benign updates as much as possible based on adjusted cosine distance, the sub-task of this attack is to try to minimize the adjusted cosine distance of malicious updates from that of benign updates. Formally, a malicious client i first derives benign and malicious updates ($\boldsymbol{w}_t^i$ and $\boldsymbol{w}_t^{i'}$) with owned unpoisoned and poisoned data ($\mathcal{D}_i$ and $\mathcal{D}_i^{'}$), respectively, at the t-th round:

$$\boldsymbol{w}_t^i \leftarrow \boldsymbol{w}_{t-1} - \eta\nabla\mathcal{L}(\boldsymbol{w}_{t-1}, \mathcal{D}_i), \boldsymbol{w}_t^{i'} \leftarrow \boldsymbol{w}_{t-1} - \eta\nabla\mathcal{L}(\boldsymbol{w}_{t-1}, \mathcal{D}_i^{'}).$$

Then, client i solves the optimization problem:

$$\arg\min_{w_t^{i'}} \lambda \mathcal{L}_i(w_{t-1}, \mathcal{D}_i^{'}) + (1-\lambda)\|w_t^i - w_t^{i'}\|_{COS},$$

where $\|\cdot\|_{COS}$ refers to adjusted cosine distance. $\lambda \in (0,1]$ is a hyperparameter to balance the efficacy and stealthiness of an attack. A smaller λ makes the attack harder to be filtered, but its efficacy is less to be upheld. Section 5.1 gives a detailed analysis.

5 EVALUATION

We use MNIST [38] and FMNIST [61] to train CNN same with [15] and CIFAR-10 [35] to train ResNet-18 [30]. Please refer to Appendix H.1 for a detailed description. To conduct a fair comparison against existing Byzantine-robust methods, we follow the training settings of [15, 49]. Based on the number of classes L, the clients are divided into L groups. Non-iid degree q determines the heterogeneity of data distribution. For example, if we use MNIST with 10 classes and $q = 0.5$, the samples with label "0" are allocated to the group "0" with probability 0.5 (but to other groups with probability $\frac{1-0.5}{10-1}$).

Byzantine-attacks settings. We consider six poisoning attacks aforementioned in Section 2.1. For GA, Krum, and Trim attacks, we adopt the default settings in [23]. To achieve a fair comparison, we follow the settings of BA [49], where a white rectangle with size 6×6 is seen as a trigger embedded on the left side of the image. The Poisoning Data Rate (PDR) is also aligned with the settings of [49]. Wang *et al.* [**?**] did not provide a dataset for FMNIST. In the experiments, we do not consider launching EA to FMNIST. To balance the main and attack tasks, we set λ as 0.5.

Dataset	MNIST	FMNIST	CIFAR-10
#clients	[10, 100, 500]		
clients subsampling rate	1		
non-iid degree	[0.1, 0.5, 0.9]		
#local epochs	1		
#global epochs	250		1200
learning rate	0.01	0.01 with $1e^{-5}$ weight decay	
proportion of malicious clients ξ	[0.1, 0.6, 0.9]		
λ	[0.1, 0.5, 1.0]		
α	1		
#edge-case	300	/	300
DP's $(\epsilon, \delta, \Delta)$	(5, $1e^{-5}$, 5)		

Table 2. FL system settings. The parameters' range and default values are in the form of "[min, default, max]".

FL system settings. Table 2 gives the detailed parameters. We follow the parameters setting of [6, 43], set the minibatch size to 128, and use the Adam optimizer [34] for training LeNet and ResNet-18. In the experiments, all the clients participate in the training from beginning to end. By default, we assume that there exist 100 clients splitting the training data with non-iid degree q=0.5; the proportion of malicious clients is set to ξ=0.6. We inject triggers into the whole testing dataset to inspect the ASR of BA. The Ardis and Southwest airplanes datasets with changed labels are used to inspect the ASR of EA in MNIST and CIFAR-10, respectively. In the clustering and robustness comparison, we define `weights-MUDGUARD` as a variant of `MUDGUARD`, which uses SGD to update models and takes pairwise adjusted cosine similarity of updates as inputs and L_2 norm as clustering metric, without applying any security tools.

Attacks		baseline	GA	LFA	Krum	Trim	AA	BA	EA
ξ	0.5	0.975	0.973	0.967	0.955	0.965	0.979 / 0	0.972 / 0.002	0.966 / 0.03
	0.6	0.977	0.975	0.974	0.952	0.96	0.979 / 0.002	0.968 / 0.001	0.968 / 0.023
	0.7	0.975	0.971	0.971	0.956	0.953	0.977 / 0	0.963 / 0.002	0.953 / 0.07
	0.8	0.969	0.968	0.964	0.942	0.944	0.976 / 0.003	0.961 / 0.005	0.965 / 0.085
	0.9	0.969	0.968	0.968	0.943	0.937	0.971 / 0.005	0.963 / 0.002	0.963 / 0.093
n	10	0.978	0.978	0.965	0.961	0.962	0.976 / 0	0.976 / 0	0.975 / 0
	50	0.975	0.97	0.958	0.96	0.949	0.975 / 0	0.975 / 0	0.967 / 0.02
	100	0.977	0.975	0.974	0.952	0.96	0.979 / 0.002	0.968 / 0.001	0.968 / 0.023
	200	0.962	0.962	0.948	0.951	0.943	0.963 / 0.002	0.961 / 0	0.962 / 0.042
	500	0.763	0.762	0.72	0.722	0.735	0.738 / 0.004	0.762 / 0.001	0.756 / 0.007
q	0.1	0.976	0.975	0.978	0.975	0.975	0.978 / 0	0.975 / 0.003	0.976 / 0.031
	0.3	0.974	0.973	0.974	0.966	0.972	0.98 / 0	0.978 / 0.002	0.978 / 0.026
	0.5	0.977	0.975	0.974	0.952	0.96	0.979 / 0.002	0.968 / 0.001	0.968 / 0.023
	0.7	0.898	0.894	0.872	0.887	0.906	0.89 / 0.013	0.876 / 0.011	0.883 / 0.039
	0.9	0.709	0.682	0.705	0.694	0.689	0.689 / 0.017	0.707 / 0.025	0.72 / 0.06

Table 3. Comparison of accuracy with baseline and ASR by an increasing proportion of malicious clients ($\xi \geq 0.5$), #clients n and non-iid degree q, where MNIST is used. The results under targeted attacks are in the form of "testing accuracy / ASR".

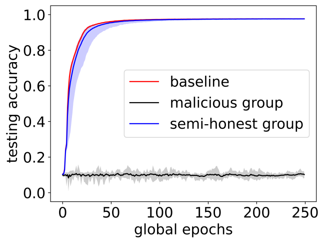
5.1 Evaluation on Accuracy

We set the baseline as a "no-attack-and-defense" FL, which means it excludes the use of any cryptographic tools as well as Byzantine-robust solutions but only trains with fully honest parties. This reaches the highest accuracy and fastest convergence speed for FL training. We then set #clients participating in the baseline training equal to the number of semi-honest clients in the malicious existence case. We conduct each experiment for 10 independent trials and further calculate the average to achieve smooth and precise accuracy performance. We evaluate MUDGUARD's accuracy and ASR by varying the total number of clients, the proportion of malicious clients, and the degree of non-iid; and further compare the performance with the baseline.

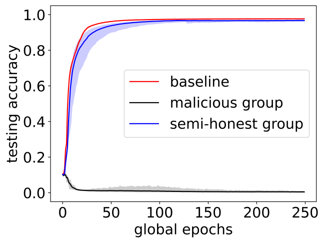
Table 3 shows that, under GA, AA, BA, and EA, the testing accuracy is on par with the baseline (with only a 0.008 gap on average) in MNIST. However, compared with the baseline, the results of MUDGUARD under LFA, Krum, and Trim attacks show slight drops (on average, 0.025 in MNIST). This is so because MUDGUARD has slow convergence and large fluctuation. This is incurred by two factors. To reduce the overheads of secure computations, we apply binary SS in SignSGD. SignSGD could cause negative impacts on clustering. Only taking the signs of the gradients can ignore the effect of the magnitudes of the malicious gradients. This makes the clustering a bit prone to inaccuracy. The other factor is that the LFA and Krum/Trim attacks either poison the training data and further poison updates or the local model to optimize the attacks. In the early stage of training, the malicious models do not perfectly fit the poisoned training data and local models yet. Thus, the semi-honest and malicious clients could be classified into the same cluster.

Figure 2 presents an overview of the testing accuracy (of baseline and semi-honest and malicious groups) and ASR (of the two groups) under Byzantine attacks in the default settings of Table 2, where MNIST is used.[3] We see that semi-honest clients can obtain comparable accuracy to the baseline at the end of the training. In Figure 2a-d, the accuracy of the semi-honest group and the baseline sharply increase from 0.1 at epoch 0 to around 0.95 at epoch 25, then gradually converge to 0.97. In the GA, since the malicious group can only receive aggregation of noise, their accuracy always fluctuates around 0.1, equalling a random guess probability. As for LFA, the model accuracy

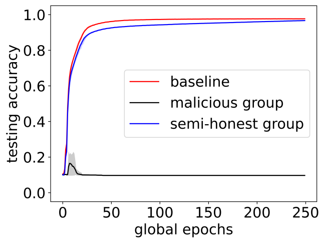
[3]The lines refer to average cases, while the shadow outlines the max and min accuracy of each epoch.



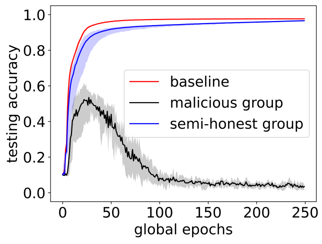
(a) Gaussian Attack



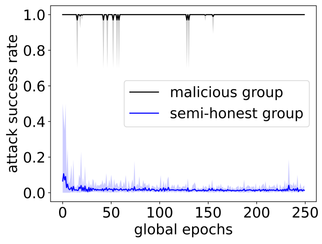
(b) Label Flipping Attack



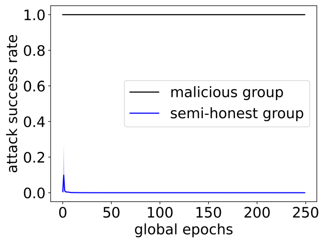
(c) Krum Attack



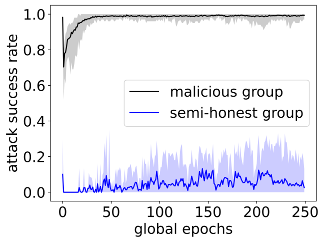
(d) Trim Attack



(e) Adaptive Attack



(f) Backdoor Attack



(g) Edge-case Attack

Fig. 2. Comparison of testing accuracy among baseline, semi-honest, and malicious groups under untargeted attacks (a-d) and ASR between the groups under targeted attacks (e-f), where we train MNIST by the default settings in Table 2.

gradually drops from 0.1 (at the beginning) to 0. This is because their models are trained on label-flipped datasets, while the labels of the testing set are not flipped. If the testing set is used to detect a poisoned model, the result should be flipped labels and failing to match the labels in the testing set, which results in 0. Since semi-honest and malicious clients can be classified into the same cluster at the beginning of the training, the accuracy of their models, w.r.t. malicious clients, is larger than 0.1 in some trials.

As shown in Figure 2b-e, the accuracy of the semi-honest group under these attacks converges slightly slower than the baseline. LFA, Krum, and Trim attacks aim to either train poisoned data or optimize local poisoned models to deteriorate the global model's testing accuracy. Due to the attacks being relatively slow and not as direct as GA, malicious updates cannot deviate 100% from benign updates at the beginning of the training (which means that malicious and semi-honest clients could be clustered together). However, with more training rounds, the deviation becomes clearer. Thus, MUDGUARD separates the two groups easily.

AA, BA, and EA have no impact on the model's testing accuracy since their main purpose is to improve the ASR (nearly equal to 1 without defense). Under MUDGUARD, the final ASR is well suppressed. The ASR of AA and BA are close to 0 in MNIST (see Figure 2f-g). However, the ASR of EA is much higher than that of AA and BA, reaching an average of 0.041. This is because, in EA, the edge-case training sets owned by attackers are very similar to the training sets with the target labels. If the discriminative capability of the model is not strong enough, the update directions of semi-honest and malicious gradients are also very close, making it difficult for MUDGUARD to distinguish them. The experimental results in FMNIST, CIFAR-10 and Shakespeare dataset [12] show the same trends as those in MNIST under the tested attacks. Due to space limitations, we present these results in Appendix H.2.

Impact of the proportion of malicious clients. We evaluate testing accuracy and ASR when the proportion of malicious clients $\xi \geq 0.5$. In Tables 3, 7, and 8, we can see that all accuracy results show a slightly downward trend with the increase of ξ in three datasets. For the baseline, the accuracy on average drops 0.008, 0.057, and 0.084 in MNIST, FMNIST, and CIFAR-10, respectively. Under GA, AA, BA, and EA, this kind of decline is on par with the baseline, whether in MNIST (0.003-0.009), FMNIST (0.059-0.66), or CIFAR-10 (0.078-0.093). Under the LFA, Krum, and Trim attacks, affected by the slow convergence and fluctuation, the testing accuracy of MUDGUARD also declines a bit more than the baseline, which is 0.012-0.028, 0.035-0.055, and 0.089-0.107 in MNIST, FMINST, and CIFAR-10, respectively. Recall that malicious clients hold a portion of the benign dataset but do not contribute to the global model (note this equals to the case where the portion of the benign dataset is missing). From this perspective, the accuracy should be related to the number of semi-honest clients, where the maximum accuracy we achieve could correspond to the case when the clients are all semi-honest. Beyond the accuracy, the ASR of EA has an upward trend while the number of malicious clients is increasing, rising by 0.005 and 0.048 in MNIST and FMNIST. Since EA is not perfectly distinguished by MUDGUARD, the ASR naturally grows with the increase in the number of malicious clients.

Impact of the total number of clients. Tables 3, 7 and 8, show the comparable testing accuracy of MUDGUARD under different attacks, as well as ASR of BA and EA when the total number of clients is set from 10 to 500. We observe that the accuracy appears to fall whilst the client number is increasing, especially when #clients = 500, it descends by about 0.2, 0.25, and 0.4 in MNIST, FMINST, and CIFAR-10, respectively. The decline in accuracy in FL is due to the reduced amount of data allocated to each client, increasing the likelihood of local models overfitting. However, MUDGUARD is not affected by this factor, and it can further defend against all untargeted attacks to maintain accuracy at the same level as the baseline. The ASR of AA and BA are controlled to nearly 0%. Although EA provides a higher ASR (than AA and BA), it drops to nearly 0 when #clients = 500, which confirms that its effectiveness relies on how well the model learns.

Impact of the degree of non-iid. We further present the testing accuracy and ASR for the cases where the degree of non-iid ranges from 0.1 to 0.9 in Tables 3, 7, and 8. We can see that in the presence of attacks, MUDGUARD can still remain at the same level of performance as the baseline, dropping only 0.018 on average. The largest decrease is 0.067 when $q = 0.5$, which happens under the Krum attack on training LeNet with FMNIST. Note the accuracy and the degree of non-iid show a negative correlation with/without attacks, which is also in line with the conclusion of [43] that FedAvg performs not well in the case of heterogeneous data distribution. The ASR of BA appears to have a slight growth as q ascends in (F)MNIST. This is because, with the high degree of non-iid, the distances among semi-honest clients also raise. For targeted attacks like AA and BA, the directions of updates are closer to those of benign updates than those of untargeted attacks. At the beginning of training, there are cases when the distances between malicious clients and semi-honest clients are similar to those between semi-honest clients, making it difficult for MUDGUARD to capture subtle differences. For the ASR of EA, as concluded in analyzing the impact of total clients, EA performs poorly when the model's accuracy is low.

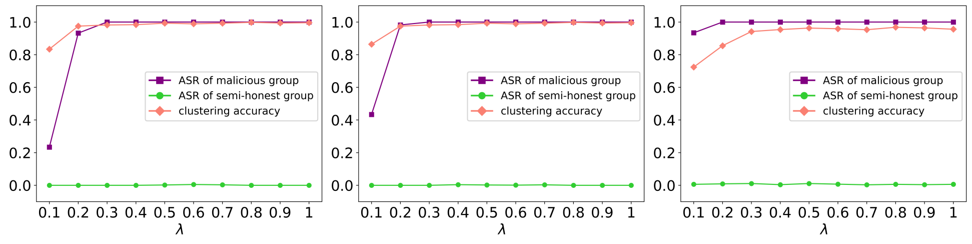
ξ =0.6		MNIST		FMNIST		CIFAR-10	
		TPR	TNR	TPR	TNR	TPR	TNR
GA	FLAME	0.821	0.846	0.848	0.847	0.879	0.928
	weights-MUDGUARD	1	1	1	1	1	1
	MUDGUARD	0.957	1	0.94	1	0.966	1
LFA	FLAME	0.653	0.612	0.634	0.655	0.742	0.711
	weights-MUDGUARD	0.974	0.987	0.975	0.977	0.98	0.985
	MUDGUARD	0.929	0.924	0.927	0.916	0.943	0.967
Krum	FLAME	0.587	0.622	0.521	0.63	0.527	0.578
	weights-MUDGUARD	0.974	0.953	0.973	0.968	0.971	0.966
	MUDGUARD	0.916	0.929	0.96	0.933	0.967	0.959
Trim	FLAME	0.691	0.679	0.699	0.664	0.646	0.615
	weights-MUDGUARD	0.976	0.964	0.975	0.965	0.973	0.988
	MUDGUARD	0.938	0.944	0.927	0.913	0.964	0.958
AA	FLAME	0.591	0.573	0.612	0.625	0.766	0.719
	weights-MUDGUARD	0.998	0.982	0.99	0.982	0.984	0.982
	MUDGUARD	0.971	0.943	0.941	0.935	0.943	0.96
BA	FLAME	0.777	0.763	0.794	0.83	0.856	0.897
	weights-MUDGUARD	0.957	0.969	0.965	0.97	0.963	0.979
	MUDGUARD	0.936	0.928	0.926	0.931	0.947	0.928
EA	FLAME	0.313	0.32	_	_	0.248	0.288
	weights-MUDGUARD	0.899	0.903	_	_	0.893	0.921
	MUDGUARD	0.856	0.876	_	_	0.827	0.83

Table 4. Effectiveness of clustering among FLAME method, weights-MUDGUARD, and MUDGUARD.

Effectiveness of clustering. To investigate the effectiveness of our clustering approach, we present the impact on True Positives Rate (TPR) and True Negatives Rate (TNR) under all attacks of $\xi = 0.6$ in Table 4 and compare against the method of FLAME.

We consider false positives to occur if semi-honest clients are grouped with the malicious. On average, under GA, the TPR and TNR improve from 0.151 and 0.126 in FLAME to 1 in weights-MUDGUARD, respectively. Since MUDGUARD is based on SignSGD, only the signs of updates are taken. Ignoring the magnitude effect, there is a reduction in TPR (an average reduction of 0.046 as compared to weights-MUDGUARD). Furthermore, TNR does not drop as we set the appropriate parameters according to Theorem 4.1. The same changes can be captured in the case of LFA: weights-MUDGUARD has an average increase of 0.3 and 0.324 in TPR and TNR, respectively, as compared to FLAME. Compared with weights-MUDGUARD, MUDGUARD drops by 0.04 and 0.05. We see that under other attacks (LFA, Krum, Trim, AA, BA, and EA), TPR and TNR are lower than the case under GA. Because they launch attacks on either training data or optimizing poisoned models, all updates at the beginning of training have high similarities, yielding those updates being clustered together and the cases of misclustering. The true rates of CIFAR-10 are higher than those of (F)MNIST, because we can set more rounds to train ResNet-18. After the model converges, the true rates reach almost 100%. Therefore, MUDGUARD obtains more correct clusters.

From the above analysis, we conclude that TNR and TPR are related to the number of training rounds, attack type, and the values of updates. Because MUDGUARD groups high similarity updates into one cluster and does not need to identify malicious/semi-honest clusters, the performance of clustering is less affected by the proportion of malicious clients. Similar results, like Table 4, can be captured even in the case when ξ >0.6. Through Figure 2, Table 4, and the above discussion, we state that although TNR and TPR are affected to a certain extent by binary SS, from the view of testing



(a) MNIST (b) FMNIST (c) CIFAR-10

Fig. 3. Impact of λ on Adaptive Attack, where backdoor attack and default settings are used.

accuracy and ASR, MUDGUARD achieves higher TPR and TNR than FLAME. Since our clustering cannot achieve 100% TPR and TNR in most cases, one may think that benign clients could be classified in the malicious clusters (i.e., always receiving malicious aggregations). However, this is not necessarily the case. Misclassification typically occurs in the early training stages before the model converges. Since the data distribution and updates between benign clients are closer than those between benign and malicious clients, early misclassifications are corrected as the model converges. The divergence between benign and malicious updates grows over time, preventing long-term misclassification. The experimental results show that our method significantly improves clustering accuracy (see Table 4), with misclassification having a sight effect on convergence speed and no impact on the final model accuracy (see Figure 2).

Impact of λ on Adaptive Attack. Figure 3 shows how the ASR varies in semi-honest and malicious groups when we adapt BA to MUDGUARD, where (F)MNIST, CIFAR-10 and default settings in Table 2 are used. In MNIST (Figure 3a), the ASR of the semi-honest group remains at a low level (nearly 0%) while that of the malicious group raises from 0.23 to 1 as λ grows from 0.1 to 0.3. This is so because when the value of λ is low, the malicious clients using AA focus more on evading filtering (i.e., inducing a drop in clustering accuracy). Even if malicious clients are grouped with semi-honest clients, they cannot produce practical attack effectiveness. When the value of λ gradually increases, the malicious clients will focus more on attack performance. In this way, MUDGUARD will easily distinguish the malicious from the semi-honest. It thus can resist AA. Note the experimental results with FMNIST and CIFAR-10 (Figure 3b and c) share the same trend with MNIST (Figure 3a). In terms of other analyses of hypeparameters (i.e., α), please refer to Appendix H.2. Appendix F shows the detailed convergence analysis of MUDGUARD.

Robustness comparison against other methods. We present a comparison among MUDGUARD and SOTA methods (FLTrust, FLAME, Zeno++, and EIFFeL) in terms of robustness, as shown in Figure 4, where MNIST is used. Several Byzantine-robust FL systems can easily and directly apply to EIFFeL. We select the two of them (please refer to [55]) for comparison, namely FLTrust and Zeno++. For brevity, we refer to them as EIFFeL-FLtrust and EIFFeL-Zeno++ hereafter. To demonstrate the advantages of MUDGUARD (based on SignSGD), we also compare its robustness with both SignSGD and FedAvg w/o defense. To investigate the impact of the cryptographic tools on testing accuracy and ASR, we also compare MUDGUARD with weights-MUDGUARD. One may see that MUDGUARD, countering the case of the malicious majority on the client side, does outperform most existing approaches.

In Figure 4a-d, the accuracy of (weights-)MUDGUARD and EIFFeL-(FLTrust/Zeno++) can be maintained at the same level as the baseline (about 0.97). Due to the impacts of misclustering, weights-MUDGUARD has a 0.02 accuracy gap with EIFFeL-FLTrust. MUDGUARD (with DP noise) commits a roughly 0.01 accuracy loss as compared to weights-MUDGUARD. The accuracy of others decreases with the increase in malicious clients, especially when $\xi \geq 0.5$, the accuracy drops abruptly to the same level of FedAvg without defense. For the ASR of AA and BA, apart from EIFFeL-(FLTrust/Zeno++), MUDGUARD and weights-MUDGUARD, all the remaining methods suddenly increase to 1 at ξ=0.4/0.5. Since EA has better attack ability (than AA and BA), weights-MUDGUARD and MUDGUARD suffer from a nearly 0.08 gap to EIFFeL-FLTrust. The ASR of others can raise from $\xi = 0.1$ and finally reach 1.0 at $\xi = 0.5$. SignSGD only limits the magnitude of malicious updates rather than filtering them out. Still, it can provide a certain level of defense (Figure 4) when there is a low malicious proportion (ξ=0.1-0.2) (compared to FedAvg having an average of 0.3 higher testing accuracy under untargeted attacks, and an average lower ASR of 0.4 under targeted attacks). As the number of malicious clients rises, its robustness drops to the level of FedAvg w/o defense.

FLAME indicates that a small-size cluster should be a malicious group. Thus, it is easy to confirm malicious clients via clustering. In the case of the malicious majority, it is hard to identify the malicious/semi-honest via group size. FLTrust assumes that before training, an honest server collects and trains on a small dataset. In each round, the server takes the updates trained by this small dataset as the root of trust. The “trusted" results are then compared to the updates sent by the clients. If the cosine similarity between them is too small, the updates will be filtered out. With this approach, the accuracy of the global model remains equivalent to that of the baseline. We state that MUDGUARD is on par with FLTrust, but it does not suffer from the restriction that the servers need to collect an auxiliary dataset ahead of training. We also see that when the proportion of malicious clients rises, the accuracy of MUDGUARD shows a slight decline. When clients upload their updates, MUDGUARD can only aggregate them with similar directions. If there is only a small percentage of semi-honest clients in the system, we naturally have an incomplete training set, causing a loss in accuracy. Note the same trends as those in MNIST can be seen in FMNIST (Figure 7) and CIFAR-10 (Figure 8). We also compare MUDGUARD with two extra Byzantine-robust FL systems in Appendix H.2.

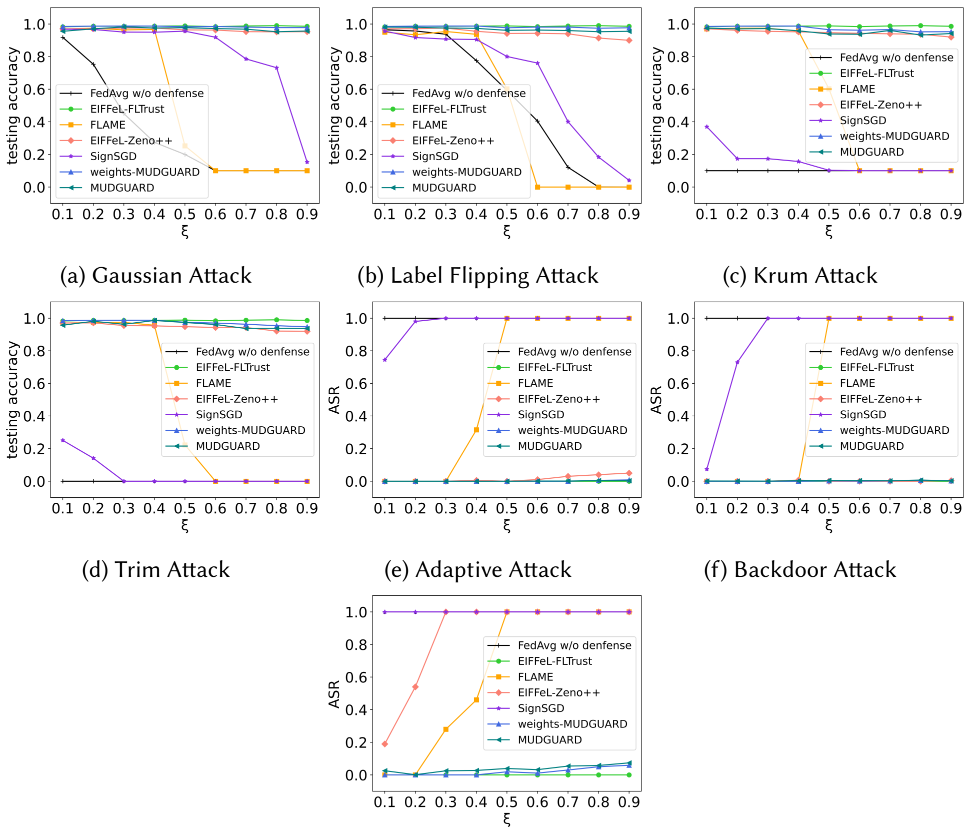
5.2 Evaluation on Overheads

Threat Model	Server-side				Client-side			
	Semi-honest		Malicious Minority		Semi-honest		Malicious Majority	
Training Model	LeNet	ResNet-18	LeNet	ResNet-18	LeNet	ResNet-18	LeNet	ResNet-18
Runtime(Second)	0.43±0.07/1.3±0.12	1.28±0.25/3.15±0.31	4.54±0.84/14.41±1.49	24.03±2.83/70.74±2.83	14.33±0.74	23.71±3.45	14.56±1.61	23.93 ±4.31
Communication Costs (MB)	16.20/53.46	34.82/314.45	873.23/2776.38	5151.18/15572.68	16.34	758.48	16.34	758.48

Table 5. Comparison of overheads among different threat models over LeNet & ResNet-18. The results on the server side are in the form of "optimized/unoptimized".

We conduct overheads assessment together with the evaluation of accuracy. The overheads presented in Table 5 capture the runtime and communication costs incurred by the implemented cryptographic tools on the server side and model training on the client side. Recall that we propose an optimization in Figure 1 (Section 4). We present the average overheads of each round of training so as to illustrate the optimized and unoptimized results in terms of different training models and honest/malicious contexts on the server side. We use LeNet and ResNet as models, and the overheads are related to their dimensionality (instead of the training data).

Runtime. In general, we see that providing robustness in the malicious context should require more runtime than in the semi-honest. This is because extra operations for verification are taken, e.g., using HHF to verify whether a received aggregation is correct. The unoptimized ResNet-18



(a) Gaussian Attack (b) Label Flipping Attack (c) Krum Attack

(d) Trim Attack (e) Adaptive Attack (f) Backdoor Attack

(g) Edge-case Attack

Fig. 4. Comparison with Byzantine-robust methods by varying ξ from 0.1 to 0.9.

takes 3.15s per round in the semi-honest context while costing 70.74s (approx. an increase of 22 times) in the malicious minority. ResNet-18 has more model parameters than LeNet, leading to extra computational operations on cryptographic tools, which can be seen, in the malicious context, 70.74s v.s. 14.41s. By binary SS and polynomial transformation, Table 5 shows that the runtime of LeNet and ResNet-18 are reduced by 68.75% (4.54s) and 66.05% (24.03s), respectively, under malicious-minority servers.

Communication costs. Similar to runtime, malicious-minority servers consume a considerable amount of communication cost compared to semi-honest ones. Table 5 shows that after optimization, the communication costs drop to 33% in the malicious minority and 10.83% in the semi-honest with ResNet-18. In the worst case, we consume 15,572.68MB bandwidth per round under malicious minority, but we optimize the cost to 5,151.18MB. In the semi-honest context, LeNet achieves the best performance, requiring 16MB with optimization, which is 30.19% of the unoptimized cost (53.46MB).

Under the same contexts, we present the overheads of the client side for FL training in Table 5. Note the use of advanced FL techniques, such as those outlined in [28, 29], can be employed to enhance computing and communication efficiency in MUDGUARD. Since applying those is straightforward, we will not go into further detail on this matter.

6 CONCLUSION

We have proposed a novel Byzantine-robust and privacy-preserving FL system. To defend against malicious majority clients, we have put introduced a new approach called *Model Segmentation* and realized it using a modified DBSCAN algorithm in which we have improved the accuracy of clustering by using pairwise cosine similarity. Leveraging cryptographic tools and DP, our design enables training to be performed correctly without breaching privacy. Our experimental results have demonstrated that the proposed protocol effectively deals with various malicious settings for both the server and client sides and outperforms most existing solutions. Our protocols introduce reasonable overheads, which we decrease by at least 3× via appropriate optimizations. In addition to theoretical and empirical analysis, we also provide extensive discussions on model accuracy, advantages of MUDGUARD, and its limitations. Due to page limits, please refer to Appendix I.

ACKNOWLEDGEMENT

This work was partly supported by the European Union's Horizon Europe Research and Innovation Program under Grant No. 101073920 (TENSOR), No. 101070052 (TANGO), No.101070627 (REWIRE) and No.101092912 (MLSysOps). Nikolaos Laoutaris was supported by the MLEDGE project (REGAGE22e00052829516), funded by the Ministry of Economic Affairs and Digital Transformation and the European UnionNextGenerationEU/PRTR.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *CCS*. 308–318.
[2] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ Compiler For Other Protocols. In *CCS*.
[3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. 2020. How to backdoor federated learning. In *AISTATS*. 2938–2948.
[4] Gilad Baruch, Moran Baruch, and Yoav Goldberg. 2019. A Little Is Enough: Circumventing Defenses For Distributed Learning. In *NIPS*.
[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *CCS*. 784–796.
[6] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed optimisation for non-convex problems. In *ICML*. 560–569.
[7] Jean-Paul Berrut and Lloyd N Trefethen. 2004. Barycentric lagrange interpolation. *SIAM review* (2004), 501–517.
[8] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. In *ICML*. 1467–1474.
[9] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS*. 118–128.
[10] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *CCS*. 1175–1191.
[11] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*. 177–186.
[12] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097* (2018).
[13] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *PAKDD*. 160–172.
[14] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*. 136–145.
[15] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. 2021. FLTrust: Byzantine-robust Federated Learning via Trust Bootstrapping. In *NDSS*.
[16] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four:Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX Security*. 2183–2200.
[17] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. 2013. Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits. In *ESORICS*. 1–18.
[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*.

[19] Ittai Dayan, Holger R Roth, Aoxiao Zhong, Ahmed Harouni, Amilcare Gentili, Anas Z Abidin, Andrew Liu, Anthony Beardsworth Costa, Bradford J Wood, Chien-Sung Tsai, et al. 2021. Federated learning for predicting clinical outcomes in patients with COVID-19. *Nature medicine* (2021), 1735–1743.
[20] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *TAMC*. 1–19.
[21] Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory* (1985), 469–472.
[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*. 226–231.
[23] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local Model Poisoning Attacks to {Byzantine-Robust} Federated Learning. In *USENIX Security*. 1605–1622.
[24] Dario Fiore, Rosario Gennaro, and Valerio Pastro. 2014. Efficiently verifiable computation on encrypted data. In *CCS*. 844–855.
[25] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*. 225–255.
[26] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. 2020. Inverting Gradients - How easy is it to break privacy in federated learning?. In *NIPS*. 16937–16947.
[27] Craig Gentry. 2009. *A fully homomorphic encryption scheme*.
[28] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. 2020. An Efficient Framework for Clustered Federated Learning. In *NIPS*. 19586–19597.
[29] Jenny Hamer, Mehryar Mohri, and Ananda Theertha Suresh. 2020. FedBoost: A Communication-Efficient Algorithm for Federated Learning. In *ICML*. 3973–3983.
[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.
[31] Tiansheng Huang, Sihao Hu, Ka-Ho Chow, Fatih Ilhan, Selim Tekin, and Ling Liu. 2024. Lockdown: backdoor defense for federated learning with isolated subspace training. *Advances in Neural Information Processing Systems* 36 (2024).
[32] Zhongzhan Huang, Wenqi Shao, Xinjiang Wang, Liang Lin, and Ping Luo. 2021. Rethinking the Pruning Criteria for Convolutional Neural Network. In *Advances in Neural Information Processing Systems*. 16305–16318.
[33] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*. 1575–1590.
[34] Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.
[35] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[36] K. Kumari, P. Rieger, H. Fereidooni, M. Jadliwala, and A. Sadeghi. 2023. BayBFed: Bayesian Backdoor Defense for Federated Learning. In *IEEE Symposium on Security and Privacy (SP)*.
[37] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* (1989), 541–551.
[38] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. (2010). http://yann.lecun.com/exdb/mnist/
[39] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure inference resilient to malicious clients. In *USENIX Security*. 2201–2218.
[40] Yehuda Lindell and Ariel Nof. 2017. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*. 259–276.
[41] Yuchen Liu, Chen Chen, Lingjuan Lyu, Fangzhao Wu, Sai Wu, and Gang Chen. 2023. Byzantine-robust learning on heterogeneous data via gradient splitting. In *ICML*. 21404–21425.
[42] Xinjian Luo, Yuncheng Wu, Xiaokui Xiao, and Beng Chin Ooi. 2021. Feature inference attack on model predictions in vertical federated learning. In *ICDE*. 181–192.
[43] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*. 1273–1282.
[44] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. 2018. The hidden vulnerability of distributed learning in byzantium. In *ICML*. 3521–3530.
[45] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *CCS*. 35–52.
[46] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*. 19–38.
[47] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE S&P*. 739–753.
[48] Milad Nasr, Reza Shokri, and Amir houmansadr. 2020. Improving Deep Learning with Differential Privacy using Gradient Encoding and Denoising. arXiv:2007.11524 [cs.LG]
[49] Thien Duc Nguyen, Phillip Rieger, Huili Chen, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Shaza Zeitouni, Farinaz Koushanfar, Ahmad-Reza Sadeghi, and Thomas Schneider. 2022. FLAME: Taming Backdoors in Federated Learning. In *USENIX Security*.

[50] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. 223–238.
[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *NIPS* (2019), 8026–8037.
[52] Michael O Rabin. 2005. How To Exchange Secrets with Oblivious Transfer. *IACR Cryptol. ePrint Arch.* (2005).
[53] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. {XONN}: Xnor-based oblivious deep neural network inference. In *USENIX Security*. 1501–1518.
[54] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security.
[55] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. 2022. EIFFeL: Ensuring Integrity for Federated Learning. In *CCS*. 2535–2549.
[56] Adi Shamir. 1979. How to share a secret. *Commun. ACM* (1979), 612–613.
[57] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *AISec*. 1–11.
[58] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
[59] Hongyi Wang, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. 2020. Attack of the Tails: Yes, You Really Can Backdoor Federated Learning. In *NIPS*. 15 pages.
[60] Nan Wu, Farhad Farokhi, David Smith, and Mohamed Ali Kaafar. 2020. The value of collaboration in convex machine learning with differential privacy. In *IEEE S&P*. 304–317.
[61] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).
[62] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. 2020. DBA: Distributed Backdoor Attacks against Federated Learning. In *ICLR*.
[63] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2020. Zeno++: Robust fully asynchronous SGD. In *ICML*. 10495–10503.
[64] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. 2018. Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates. In *ICML*. 5650–5659.
[65] Ligeng Zhu, Zhijian Liu, and Song Han. 2019. Deep leakage from gradients. *NIPS* (2019).

A NOTATION

The frequently used notations are in Table 6.

B IMPLEMENTATION ALGORITHMS

In the evaluation, we implement the proposed MUDGUARD (with optimization) mainly based on Algorithms 1 and 2.

C TOOLS

C.1 Federated Learning

Federated Learning (FL) enables n clients to train a global model $\boldsymbol{w}$ collaboratively without revealing local datasets. Unlike centralized learning, FL requires clients to upload the weights of local models ($\{\boldsymbol{w}^i \mid i \in n\}$) to a parametric server. It aims to optimize a loss function: $\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \frac{k_i}{K} \mathcal{L}_i(\boldsymbol{w}, \mathcal{D}_i)$, where $\mathcal{L}_i(\cdot)$ and k_i are the loss function and local data size of i-th client. At t-th round, the FL training can usually be divided into the following steps.

• *Global model download*: The server selects partial clients engaging in training. All connected clients download the global model $\boldsymbol{w}_{t-1}$ from the server.

• *Local training*: Each client updates its local model by training with its own dataset: $\boldsymbol{g}_{t-1}^{i} \leftarrow \frac{\partial \mathcal{L}(\boldsymbol{w}_{t-1}, \mathcal{D}_i)}{\partial \boldsymbol{w}_{t-1}}$.

• *Aggregation*: After the local updates $\{\boldsymbol{g}_{t-1}^{i} \mid i \in n\}$ are uploaded, the server updates the global model by aggregation: $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \sum_{i=1}^{n} \frac{k_i}{K} \boldsymbol{g}_{t-1}^{i}$, where η refers to the learning rate.

Notation	Description
$\boldsymbol{g}_t^i$	gradients of i-th client at t-th round
$\boldsymbol{w}_t^i$	weights of i-th client at t-th round
T	the number of rounds
n	the number of clients
S	the number of servers
m	the number of malicious clients
k_i	the number of data instances of i-th client
c	the number of clusters
l	the cluster labels
E	the number of epochs
$\mathcal{D}_i$	the dataset of i-th client
η	learning rate
$\boldsymbol{G}^z$	the aggregation of gradients of z-th cluster
$[n]$	a set of numbers ranging from 1 to n
$[[\cdot]]$	secret shared format
CosM	pairwise adjusted cosine similarity matrix
EudM	pairwise L_2 distance matrix
IndM	indicator matrix
δ, ϕ	secret keys of homomorphic hash function
Δ, σ, ϵ	parameters of differential privacy
$\mathcal{N}$	Gaussian noise
α	density parameter
ECD$(\cdot)$	encoding algorithm
DCD$(\cdot)$	decoding algorithm

Table 6. Notation summary

C.2 DBSCAN

Unlike traditional clustering algorithms (e.g., k-means, k-means++, bi-kmeans), which need to pre-define the number of clusters, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [22] is proposed to cluster data points dynamically. Based on density-based clustering, DBSCAN guarantees that clusters of any shape can always be identified. Besides, it can recognize noise points effectively. Basically, after setting the density parameter (α) and the minimum cluster size ($mPts$), DBSCAN can conduct effective clustering. We note HDBSCAN [13] could also be used for clustering. Its main difference from DBSCAN is the multiple densities clustering. In this work, we assume that malicious clients may only conduct one kind of attack during the whole training, e.g., a group of malicious clients conducting a Label Flipping Attack together. The malicious updates could only derive one density. Like [49], we may apply HDBSCAN in the clustering. However, DBSCAN, in general, requires less computational complexity than HDBSCAN in algorithmic constructions. And further, we will conduct the clustering with cryptographic operations. Considering efficiency, we choose DBSCAN over HDBSCAN.

C.3 Cryptographic Tools

The secure Multiparty Computation (MPC) framework aims to enable multiple parties to evaluate a function over ciphertexts securely. The parties conducting MPC can access inputs via protection approaches, e.g., in a secret-shared format. It does not leak any information besides the final output unless these shares are combined to derive plaintexts.

Secret Sharing (SS). It refers to a type of tool for splitting a secret among multiple parties, each of whom is assigned a share of the secret. The security of an SS scheme guarantees that one can distinguish shares and randoms with a negligible probability. Apart from that, no one can

Algorithm 1: MUDGUARD.

```
Input: training dataset $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i$
Output: global models $\{\boldsymbol{w}^i \mid i \in [n]\}$
1  ServerAggregation:
2  Initialize global model $\boldsymbol{w}_0$
3  for each global epoch $t = 1,2,\cdots,T$ do
4      for client $i \in n$ in parallel do
5          $[[\boldsymbol{g}_t^i]] \leftarrow$ ClientUpdate($i$, $\boldsymbol{w}_{t-1}^i$)
6      end
7      $[[\boldsymbol{G}_t^j]] \leftarrow$ Algorithm 2
8      return $[[\boldsymbol{G}_t^j]], \mathsf{H}_{\delta,\phi}([[\boldsymbol{G}_t^j]])$ to clients
9  end
10 ClientUpdate($i$, $\boldsymbol{w}_{t-1}^i$):
11 $\mathcal{B} \leftarrow$(split $\mathcal{D}_i$ into batches of size $b$)
12 $\mathsf{IndM} \leftarrow$MajorityVote($\{\mathsf{IndM}_i \mid i \in [s]\}$)
13 $l \leftarrow$DBSCAN($\mathsf{IndM}$)
14 reconstruct $\boldsymbol{G}_{t-1}^i$ by $[[\boldsymbol{G}_{t-1}^i]]$
15 if $\prod \mathsf{H}_{\delta,\phi}(\hat{\boldsymbol{g}}_{t-1}^i) = \mathsf{H}_{\delta,\phi}(\boldsymbol{G}_{t-1}^i)$ then
16     accept and continue
17 else
18     refuse and break
19 end
20 $\boldsymbol{w}_t^i \leftarrow \boldsymbol{w}_{t-1}^i - \eta \cdot \mathsf{sign}(\boldsymbol{G}_{t-1}^i)$
21 $\boldsymbol{g}_t^i \leftarrow$LocalTraining($\boldsymbol{w}_t^i$; $batch$; $loss$)
22 $\widetilde{\boldsymbol{g}}_t^i \leftarrow \boldsymbol{g}_t^i / \max(1, \|\boldsymbol{g}_t^i\|_2/\Delta) + \mathcal{N}(0, \Delta^2\sigma^2)$
23 $\hat{\boldsymbol{g}}_t^i \leftarrow \mathsf{KS}(\widetilde{\boldsymbol{g}}_t^i, \mathcal{N}) \cdot \widetilde{\boldsymbol{g}}_t^i$
24 $\overline{\boldsymbol{g}}_t^i \leftarrow \mathsf{ECD}(\mathsf{sign}(\hat{\boldsymbol{g}}_t^i))$
25 send $[[\overline{\boldsymbol{g}}_t^i]]$ to servers
26 broadcast $\mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{\boldsymbol{g}}_t^i))$
```

reconstruct the secret unless holding all (or a subset) of shares. Let us consider Shamir Secret Sharing (SSS) [56] (t, n)-threshold scheme as an example. Assume one chooses a polynomial $f(x) = \sum_{i=0}^{t-1} a_i \cdot x^i$ over $\mathbb{Z}_q$ and a secret $a_0 = f(0)$. The secret can be split into n shares by randomly selecting n values: $\{r_j \leftarrow \mathbb{Z}_q^* \mid j \in n\}$, and then calculating shares $\{f(r_j) \mid j \in n\}$. Given a subset of any t out of n shares, the secret can by reconstructed by Lagrange interpolation [7]:$f(0) = \sum_{j=0}^{t-1} f(r_j) \cdot \prod_{z=0,z\neq j}^{t-1} \frac{r_z}{r_z - r_j}$. Except for SSS, other schemes like additive SS and replicated SS are used in the MPC framework [17, 33]. Note these schemes have a linear property. Even if each party performs linear combinations locally with shares, the combined secret matches the result obtained by these linear calculations. This saves significant communication costs in the FL context, where servers are only required to aggregate shares of gradients.

Homomorphic Hash Functions (HHF). Given a message $x \in \mathbb{Z}_q$, a collision-resistant HHF [24] H: $\mathbb{G}_1 \times \mathbb{G}_2 \leftarrow \mathbb{Z}_q$ can be indicated as $\mathsf{H}(x) = (g^{\mathsf{H}'_{\delta,\phi}(x)}, h^{\mathsf{H}'_{\delta,\phi}(x)})$, where δ and ϕ are secret keys randomly and independently selected from $\mathbb{Z}_q$. H' is a hash function, and $\mathbb{G}_1$ and $\mathbb{G}_2$ are two different groups. Similar to other one-way hash functions, the security of the HHF requires that one can find a collision with a negligible probability. Based on additive homomorphism: $\mathsf{H}(x_1 + x_2) \leftarrow$

Algorithm 2: Secure clustering.

```
  Input: shares of gradients: ${[[\overline{g}_t^i]] \mid i \in [n]}$
  Output: shares of aggregation ${[[G_t^z]] \mid z \in c}$
1 for each $i, j \in n$ do
2     $[[dot\_product_{ij}]] \leftarrow [[\overline{g}_t^i]] \oplus [[\overline{g}_t^j]]$
3     convert binary sharing ($[[dot\_product_{ij}]]$, $[[\overline{g}_t^i]]$) to arithmetic sharing by B2A
4     $[[\mathsf{CosM}_{ij}]] \leftarrow 1 - \frac{2}{np} \sum [[dot\_product_{ij}]]$
5 end
6 for each $i, j \in n$ do
7     $[[x_{ij}]] \leftarrow ([[\mathsf{CosM}_i]] - [[\mathsf{CosM}_j]])^2$
8     $[[\mathsf{EucM}_{ij}]] \leftarrow 1 + \frac{[[x_{ij}]]-1}{2} - \frac{([[x_{ij}]]-1)^2}{8} + \frac{([[x_{ij}]]-1)^3}{16}$
9     if $\mathsf{EucM}_{ij} \leq \alpha$ then
10        $[[\mathsf{IndM}_{ij}]] == [[1]]$
11    else
12        $[[\mathsf{IndM}_{ij}]] == [[0]]$
13    end
14 end
15 reconstruct IndM
16 each server broadcasts IndM
17 $l \leftarrow$ DBSCAN(IndM)
18 for each $z \in c$ all servers in parallel do
19     $[[G_t^z]] \leftarrow \sum_{l_i=z} \mathsf{DCD}([[\overline{g}_t^i]]), i \in n$
20     return $[[G_t^z]]$ to clients ${i \mid l_i = z}$
21 end
```

$(g^{\mathsf{H}'_{\delta,\phi}(x_1)+\mathsf{H}'_{\delta,\phi}(x_2)}, h^{\mathsf{H}'_{\delta,\phi}(x_1)+\mathsf{H}'_{\delta,\phi}(x_2)})$, in this work, we will use this tool as a verification of the correctness of aggregation.

Homomorphic Encryption (HE). This tool is an interesting privacy-preserving technology that enables users to evaluate polynomial computations on ciphertexts without revealing underlying plaintexts. An encryption scheme is called partial HE if it only supports addition [50] or multiplication [21], while fully HE [27] can support both. An HE scheme usually includes the following steps.

• *Key Generation:* $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$, where based on security level parameter λ, public key pk and secret key sk can be generated.

• *Encryption:* $(c_1, c_2) \leftarrow \mathsf{Enc}(\mathrm{pk}, m_1, m_2)$. By using pk, the probabilistic algorithm encrypts messages m_1, m_2 to ciphertexts c_1, c_2.

• *Homomorphic evaluation:* $\mathsf{Eval}(c_1, c_2) = c_1 \circ c_2 = \mathsf{Enc}(\mathrm{pk}, m_1) \circ \mathsf{Enc}(\mathrm{pk}, m_2) = \mathsf{Enc}(\mathrm{pk}, m_1 \circ m_2)$, where $\circ$ refers to an operator, e.g., addition or multiplication.

• *Decryption:* $m_1 \circ m_2 \leftarrow \mathsf{Dec}(\mathrm{sk}, \mathsf{Enc}(\mathrm{pk}, m_1 \circ m_2))$. Using sk, the operational results of m_1 and m_2 can be derived.

Oblivious Transfer (OT). OT [52] is one of the crucial building blocks for MPC. In an OT protocol (involving two parties), a sender holds n different strings $s_i, i = 1 \cdots n$, and a receiver has an index (ind) and wants to learn s_{ind}. At the end of the protocol, the receiver cannot get information about strings rather than s_{ind}, while the sender learns nothing about ind selected by the receiver. For example, a 1-out-of-2 OT protocol only inputs two strings and a 1-bit index.

Garbled Circuits (GC) [5]. The protocol is run between two parties called the garbler and evaluator. The garbler generates the GC corresponding to the Boolean circuit to be evaluated securely by associating two random keys per wire representing the bit values 0, 1. The garbler then sends the GC together with the keys for the inputs to the evaluator. The evaluator obliviously obtains the keys for his inputs via OT and evaluates the circuit to obtain the output key. Finally, the evaluator maps the output key to the real output.

C.4 Differential Privacy

Differential Privacy (DP) [20] is a data protection approach enabling one to publish statistical information of datasets while keeping individual data private. The security of DP requires that adversaries cannot statistically distinguish the changes between two datasets where an arbitrary data point is different. The most widely used DP mechanism is called (ϵ, δ)-DP defined below, requiring less injection noise than the ϵ-DP but standing at the same privacy level.

Definition C.1 ((ϵ, δ) - Differential Privacy [20]). Given two real positive numbers (ϵ, δ) and a randomized algorithm $\mathcal{A}$: $\mathcal{D}^n \to \mathcal{Y}$, the algorithm $\mathcal{A}$ provides (ϵ, δ) - DP if for all data sets $\mathbf{D}, \mathbf{D}' \in \mathcal{D}^n$ differing in only one data sample, and all $\mathcal{S} \subseteq \mathcal{Y}$: $Pr[\mathcal{A}(\mathbf{D}) \in \mathcal{S}] \leq exp(\epsilon) \cdot Pr[\mathcal{A}(\mathbf{D}') \in \mathcal{S}] + \delta$.

Note that the Gaussian noise $\mathcal{N} \sim N(0, \Delta^2\sigma^2)$ should be added to the output of the algorithm, where Δ is L_2 sensitivity of $\mathbf{D}$ and $\sigma = \sqrt{2\ln(1.25/\delta)}$ [1]. The robustness of post-processing guarantees for any probabilistic/deterministic functions $\mathcal{F}$, if $\mathcal{A}$ satisfies (ϵ, δ)-DP, so does $\mathcal{F}(\mathcal{A})$.

D COMPLEXITY ANALYSIS

We use d to denote the dimension of the model. n, S, and c refer to the number of clients, servers, and clusters, respectively.

• **Computation cost.** Each client's computation cost can be computed as binary SS with Tiny OT $- O(d)$. The server's computation cost consists of 4 parts: (1) computing pairwise XOR $- O(n^2)$; (2) bit to arithmetic conversion $- O(d)$; (3) multiplication for L_2 distance $- O(n^3)$; (4) comparison with $\alpha - O(n^2)$; (5) calculating results of HHF based on the number of clusters $c - O(nc)$. Thus, the total computation complexity of each server is $O(n^3)$.

• **Communication cost.** For a client in *MUDGUARD*, the communication cost can be divided into two parts: (1) sending updates to S servers with binary SS and Tiny OT $- O(Sd)$; (2) broadcasting hash results of updates to the rest of parties $- O(n + S)$. Thus, we have communication complexity $- O(Sd + n)$ for each client. The servers' communication costs include (1) receiving correlated randomness and doubly-authenticated bits for converting a boolean shared matrix to arithmetic one $- O(n^2)$; (2) receiving triples for multiplications $- O(n^2)$; (3) receiving correlated randomness for element-wise comparison $- O(n^2)$; (4) sending shares and a random bit to other servers for reconstruction $- O(Sn^2)$; (5) sending aggregated shares and values of HHF to all clients $- O(nd)$. Overall, the communication cost for every server is $O(Sn^2)$. For detailed experimental results, refer to Section 5.2.

E SECURITY ANALYSIS

We first define the ideal functionality $\mathcal{F}_{\text{MUDGUARD}}$ to execute a byzantine-robust privacy-preserving FL, and then show that the proposed protocol MUDGUARD securely realizes the functionality. Our security is based on the random oracle model, where the homomorphic hash function outputs a uniformly random value for a new query and the same value for a previously answered query. Hence, we prove the UC security in $\mathcal{F}_{\text{RO}}$-hybrid model. Besides, the security is also based on the existence of a secret sharing protocol where the clients derive shares indistinguishable with randoms, which securely realizes $\mathcal{F}_{\text{SS}}$ (a combination of $\mathcal{F}_{\text{triples}}, \mathcal{F}_{\text{share}}, \mathcal{F}_{\text{reconst}}$ [25]), and a bit-to-arithmetic conversion protocol that securely realizes $\mathcal{F}_{\text{B2A}}$ (noted as $\mathcal{F}_{\text{PREP}}$ in [54]).

Ideal Functionality $\mathcal{F}_{\mathsf{share}}$

The functionality $\mathcal{F}_{\mathsf{share}}$ interacts with a dealer party P_j, and a corrupted party P_i.

Upon receiving (t_i, s_i) from the corrupted party P_i, and receiving v from the dealer P_j, the functionality $\mathcal{F}_{\mathsf{share}}$ computes (t_{j+1}, s_{j+1}) and (t_{j+2}, s_{j+2}) from (t_i, s_i) and v, and sends the honest P_{i-1} and P_{i+1} their respective shares.

Ideal Functionality $\mathcal{F}_{\mathsf{reconst}}$

The functionality $\mathcal{F}_{\mathsf{reconst}}$ interacts with an adversary Sim and a corrupted party P_i, and receives information from P_{i+1} and P_{i+2}.

Upon receiving (t_{i+1}, s_{i+1}, j) from P_{i+1} and (t_{i+2}, s_{i+2}, j) from P_{i+2}, $\mathcal{F}_{\mathsf{reconst}}$ computes $v = s_{i+2} \oplus t_{i+1}$ and sends v to P_j. In addition, the functionality $\mathcal{F}_{\mathsf{reconst}}$ sends (t_i, s_i) to the adversary Sim, where (t_i, s_i) is P_i's share as defined by the shares received from the honest parties.

Ideal Functionality $\mathcal{F}_{\mathsf{triples}}$

The functionality $\mathcal{F}_{\mathsf{triples}}$ interacts with a corrupted party P_i, and receive information from $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$.

Upon receiving N triples of pairs $\{(t^j_{a_i}, s^j_{a_i}), (t^j_{b_i}, s^j_{b_i}), (t^j_{c_i}, s^j_{c_i})\}_{j=1}^N$ from P_i, the functionality first $\mathcal{F}_{\mathsf{triples}}$ chooses random $a_j, b_j \in \{0,1\}$ and computes $a_j b_j$, and then defines a vector of sharings $\boldsymbol{d} = ([a_j], [b_j], [c_j])$, for $j = 1, ..., N$. The sharings are computed from $[(t^j_{a_i}, s^j_{a_i}), (t^j_{b_i}, s^j_{b_i}), (t^j_{c_i}, s^j_{c_i})]$ provided by P_i and the chosen a_j, b_j, c_j. Next, $\mathcal{F}_{\mathsf{triples}}$ sends the generated shares to each corresponding party.

Ideal Functionality $\mathcal{F}_{\mathsf{Prep}}$ ($\mathcal{F}_{\mathsf{B2A}}$)

Independent copies of $\mathcal{F}_{\mathsf{MPC}}$ are identified via session identifiers sid. For each instance, $\mathcal{F}_{\mathsf{Prep}}$ maintains a dictionary $\mathsf{Dic}_{\mathsf{sid}}$. If a party provides input with an invalid sid, the $\mathcal{F}_{\mathsf{Prep}}$ outputs **reject** to all parties and await another message.

Upon receiving (**Init**, $\mathbb{F}$, sid) from all parties, initialize a new database of secrets $\mathsf{Dic}_{\mathsf{sid}}$ indexed by a set $\mathsf{Dic}_{\mathsf{sid}}$.Keys and store the field $\mathbb{F}$as $\mathsf{Dic}_{\mathsf{sid}}$.Field, if sid is a new session identifier. Set the flag $\mathsf{Abort}_{\mathsf{sid}} = \textsc{false}$.

Upon receiving (**Input**, i, id, x, sid) from a party P_i and (**Input**, i, id, $\perp$, sid) from all other parties, if id $\notin \mathsf{Dic}_{\mathsf{sid}}$.Keys then insert it and set $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}] = x$. Then execute the procedure **Wait**.

Upon receiving (**Add**, $\mathsf{id}_x, \mathsf{id}_y$, id, sid), set $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}] = \mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}_x] + \mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}_y]$ if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Dic}_{\mathsf{sid}}$.Keys.

Upon receiving (**Mult**, $\mathsf{id}_x, \mathsf{id}_y$, id, sid), set $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}] = \mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}_x] \cdot \mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}_y]$, if $\mathsf{id}_x, \mathsf{id}_y \in \mathsf{Dic}_{\mathsf{sid}}$.Keys. Then execute the procedure **Wait**.

Upon receiving (**RanEle**, id, sid), set $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}]$ to a random element in $\mathsf{Dic}_{\mathsf{sid}}$.Field, if id $\notin \mathsf{Dic}_{\mathsf{sid}}$.Keys. Then execute the procedure **Wait**.

Upon receiving (**RanBit**, id, sid), set $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}]$ to a random bit if id $\notin \mathsf{Dic}_{\mathsf{sid}}$.Keys. Then execute the procedure **Wait**.

Upon receiving (**Open**, i , id, sid) from all parties, if id $\in$ $\mathsf{Dic}_{\mathsf{sid}}$.Keys: 1) if $i = 0$, send $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary and executes **Wait**. If the answer is (**OK**, sid), await an error ϵ from the adversary. Send $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}] + \epsilon$ to all honest parties. If $\epsilon \neq 0$, set the flag $\mathbf{Abort}_{\mathsf{sid}}$ = TRUE. 2) if $i \in A$, then send $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}]$ to the adversary. Then execute **Wait**. 3) if $i \in [n] \backslash A$, execute **Wait**. If not already halted, then await an error ϵ from the adversary. Send $\mathsf{Dic}_{\mathsf{sid}}[\mathsf{id}] + \epsilon$ to party P_i. If $\epsilon \neq 0$, set the flag $\mathbf{Abort}_{\mathsf{sid}}$ = TRUE.

Upon receiving (**Check**, sid) from all parties, execute the procedure **Wait**. If not already halted and $\mathbf{Abort}_{\mathsf{sid}}$ = TRUE, send (**Abort**, sid) to the adversary and all honest parties, and ignore further messages to $\mathcal{F}_{\mathsf{MPC}}$ with the same sid. Otherwise, send (**OK**, sid) and continue.

Upon receiving (**daBits**, $\mathsf{id}_1, ..., \mathsf{id}_l, \mathsf{sid}_1, \mathsf{sid}_2$) from all parties where $\mathsf{id}_i \notin \mathsf{Dic}_{\mathsf{sid}}$.Keys for all $i \in [l]$, await a message **OK** or **Abort** from the adversary. If **OK** is received, sample a set of random bit $\{b_j\}_j \in [l]$, and for each $j \in [l]$ set $\mathsf{Dic}_{\mathsf{sid}_1}[\mathsf{id}_j] = b_j$ and $\mathsf{Dic}_{\mathsf{sid}_2}[\mathsf{id}_j] = b_j$, and insert the set $\{\mathsf{id}_i\}_{i\in[l]}$ into $\mathsf{Dic}_{\mathsf{sid}_1}$.Keys and $\mathsf{Dic}_{\mathsf{sid}_2}$.Keys. Otherwise, send (**Abort**, sid_1) and (**Abort**, sid_2) to the adversary and all honest parties, and ignore all further messages to $\mathcal{F}_{\mathsf{MPC}}$) with the same sid_1 and sid_2.

Procedure **Wait**: Await a message (**OK**, sid) or (**Abort**, sid) from the adversary. If **OK** is received, then continue. Otherwise, send (**Abort**, sid) to all honest parties, and ignore all further messages to $\mathcal{F}_{\mathsf{MPC}}$ with the same sid.

REMARK. *We use $\mathcal{F}_{share}$ as the secret share generating algorithm, $\mathcal{F}_{reconst}$ as the reconstructing algorithm, $\mathcal{F}_{triples}$ as the secret share multiplication algorithm, and $\mathcal{F}_{B2A}$ (see $\mathcal{F}_{PREP}$ in [54]) as the bit to arithmetic conversion algorithm.*

We formally define $\mathcal{F}_{\mathsf{MUDGUARD}}$ as follows.

Ideal Functionality $\mathcal{F}_{\mathsf{MUDGUARD}}$

The functionality $\mathcal{F}_{MUDGUARD}$ is parameterized with a DBSCAN algorithm with corresponding parameters, a local training SGD algorithm with appropriate variables, Gauss noise parameters Δ and σ, and the density parameter α. The functionality $\mathcal{F}_{MUDGUARD}$ interacts with n clients $\mathsf{P}_1, ..., \mathsf{P}_n$, s remote servers $\mathsf{S}_1, ..., \mathsf{S}_s$, and an ideal adversary Sim.

Upon receiving (**Init**, $\{w_0^i\}_{i\in[n]}, \{\boldsymbol{G}_0^i\}_{i\in[n]}$) from the adversary Sim, send (**Init**, $w_0^i, \boldsymbol{G}_0^i$) to each P_i.

Upon receiving (**Update**, $t, w_{t-1}^i, \mathcal{D}_i$) from each honest client P_i, calculate $w_t^i, \boldsymbol{g}_t^i, \tilde{\boldsymbol{g}}_t^i, \hat{\boldsymbol{g}}_t^i, \bar{\boldsymbol{g}}_t^i$, and store ($t$, $[[\bar{\boldsymbol{g}}_t^i]]$) for each server, and notify Sim with (**Update**, t, P_i). If $t = T$, terminate the protocol. Later, when Sim replies with (**Update-data**, t), send (**Update**, t) to each server S_j for each $j \in [s]$. Upon receiving (**Update**, t, $\{[[\bar{\boldsymbol{g}}'^i_t]]\}_{i\in\mathcal{I}}$) from Sim for all corrupted client index i, where $\mathcal{I} \subset [n]$, store (t, $[[\bar{\boldsymbol{g}}'^i_t]]$) for each honest server.

Upon receiving (**Update**, t) from a server S_j, if (t, $[[\bar{\boldsymbol{g}}_t^i]]$) is stored for each $i \in [n]$ and for each server, calculate IndM and $\{[[\boldsymbol{G}_t^z]]\}_{z\in[c]}$ for each server. Then send (**Model**, t, IndM, $\{[[\boldsymbol{G}_t^z]]\}_{z\in[c]}$) to each server. If S_j is honest, upon receiving (**Update-model**, t) from the simulator Sim, send (**Update-model**, t, $[[\boldsymbol{G}_t^z]]$) to the corresponding client. Otherwise, upon receiving (**Update-model**, t, $\{[[\boldsymbol{G}'^z_t]]\}$) from the simulator Sim, send (**Update-model**, t, $[[\boldsymbol{G}'^z_t]]$) to corresponding client.

Upon receiving (**Abort**) from either the adversary or any client, send $\perp$ to all parties and terminate.

Remark. *According to the ideal functionality, we could capture not only privacy but also soundness against malicious corruption of servers. However, the differential attack is based on the output of each epoch, which is published roundly and could be obtained legally. Hence, the discussion on differential privacy is not included in this security definition. Detailed proof of differential privacy will be provided later. Moreover, soundness against malicious corruption of clients is also not captured by the previous definition since such security is protected by the clustering technique, which is not the concern of cryptography.*

Definition E.1 (Universally Composable security). A protocol Π UC-realizes ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that, for any PPT environment $\mathcal{E}$, the ensembles $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ and $\mathsf{EXEC}_{\mathsf{IDEAL}_{\mathcal{F}},\mathcal{S},\mathcal{E}}$ are indistinguishable.

Definition E.2 (UC security of MUDGUARD*).* A protocol $\Pi_{MUDGUARD}$ is UC-secure if $\Pi_{MUDGUARD}$ UC-realizes $\mathcal{F}$, against malicious-majority clients and malicious-minority servers, considering arbitrary collusion between malicious parties.

Theorem E.3 (UC security of MUDGUARD). *Suppose the existence of a homomorphic hash function in a random oracle model, our protocol is UC-secure in $(\mathcal{F}_{RO}, \mathcal{F}_{SS}, \mathcal{F}_{B2A})$-hybrid world.*

Proof. We show the validity of the theorem by proving that the protocol Π_{MUDGUARD} securely realizes $\mathcal{F}$ in the $(\mathcal{F}_{\mathrm{RO}}, \mathcal{F}_{\mathrm{SS}}, \mathcal{F}_{\mathrm{B2A}})$-hybrid world against any corruption pattern. We construct a simulator Sim for any non-uniform PPT environment $\mathcal{E}$ such that $\mathsf{EXEC}^{\mathcal{F}_{\mathrm{RO}},\mathcal{F}_{\mathrm{SS}},\mathcal{F}_{\mathrm{B2A}}}_{\mathsf{MUDGUARD},\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{F}_{\mathsf{MUDGUARD}},\mathsf{Sim},\mathcal{Z}}$. The Sim is constructed as follows.

It writes on $\mathcal{A}$'s input tape upon receiving an input value from $\mathcal{E}$, as if coming from $\mathcal{E}$, and writes on $\mathcal{Z}$'s output tape upon receiving an output value from $\mathcal{A}$, as if from $\mathcal{A}$.
Case 1: If all clients and servers are not corrupted. Since we assume private channels between client and server, Sim could just simply randomly choose all intermediate values. There is no distinguisher who could tell the difference between random values and real transcripts.
Case 2: If corrupted clients exist. We note the corrupted subset as $\mathcal{I} \subset [n]$. We need to simulate the adversary's view, which is the secret share and its corresponding hash value. For $t \in [T]$ and $i \in \mathcal{I}$, the simulator randomly chosen $\bar{g}'^{i}_{t} \leftarrow \{0,1\}$, and internally executes $\mathcal{F}_{\mathrm{SS}}$ to obtain $[[\bar{g}'^{i}_{t}]]$. Then, Sim internally executes $\mathcal{F}_{\mathrm{RO}}$ to obtain H^{i}_{t}. Because of the UC security of $\mathcal{F}_{\mathrm{SS}}$ and $\mathcal{F}_{\mathrm{RO}}$, there is no distinguisher that could tell the difference between $([[\bar{g}'^{i}_{t}]], \mathsf{H}^{i}_{t})$ and $([[\bar{g}^{i}_{t}]], \mathsf{H}_{\delta,\phi}(\mathsf{sign}(\hat{g}^{i}_{t})))$.
Case 3: If corrupted servers exist. We not the corrupted subset as $\mathcal{I} \subset [T]$. We need to simulate the adversary's view, including all secret shares in the protocol. It is worth noticing that we should not only guarantee the indistinguishability between two groups of shares but also the relationship among elements within each group. After obtaining IndM, the Sim executes DBSCAN protocol on IndM and acquires cluster labels l, and executes the functionality $\mathcal{F}_{\mathrm{SS}}$ to obtain $[[\mathsf{IndM}_{ij}]]$ for each $i, j \in [n]$. Then, Sim randomly chosen $|l|$ secret sharing values $[[\bar{g}'^{i}_{t}]]$ such that the summation $\Sigma_{l_i=z}\mathsf{DCD}([[\bar{g}'^{i}_{t}]])$ equals to the given share $[[\boldsymbol{G}^{z}_{t}]]$. This procedure could be easily achieved by first randomly choosing the first $|l|-1$ values and then calculating the last value. For each $i \notin c$, the simulator Sim simply chooses the shares of gradients randomly since those values are irrelevant to the calculation. After acquiring all the shares of gradients $[[\bar{g}'^{i}_{t}]]$, the simulation Sim pairwisely calculate $[[dot\_product'_{ij}]]$, and convert it to arithmetic sharing by executing the functionality $\mathcal{F}_{\mathrm{B2A}}$, and then calculate the adjusted cosine similarity matrix share $[[\mathsf{CosM}'_{ij}]]$. Next, as in Algorithm 2, Sim calculates $[[x'_{ij}]]$, $[[\mathsf{EucM}'_{ij}]]$ for each $i, j \in [n]$. We claim that all shares that were previously generated are interdeducible, except between $[[\mathsf{EucM}'_{ij}]]$ and $[[\mathsf{IndM}'_{ij}]]$, since the latter two are the input/output pair of the element-wise comparison algorithm computed by a secure comparison algorithm in our protocol. Fortunately, the privacy of a secure comparison algorithm guarantees

the indistinguishability between real and ideal input/output pairs. Hence, we claim that if there exists a distinguisher that could tell the difference between the real and ideal world, it contradicts either the UC security of $\mathcal{F}_{\mathrm{SS}}$ or the privacy of secure comparison protocol.

Case 4: If there exists both corrupted clients and servers. The situation, in this case, is simply the combination of Case 2 and 3 since there is no extra view needed to simulate.

In summary, for any PPT adversary $\mathcal{A}$ we could construct a Sim, so that for any PPT environment $\mathcal{E}$, the $\mathsf{EXEC}_{\mathsf{MUDGUARD},\mathcal{A},\mathcal{E}}$ and $\mathsf{EXEC}_{\mathcal{F}_{\mathsf{MUDGUARD}},\mathcal{S},\mathcal{E}}$ are indistinguishable. □

UC framework captures attacks on input and intermediate data. On the contrary, differential privacy prevents the adversary from inferring about private information from outputs or updates, and such information might also be utilized by malicious clients. When false positives clustering exists, or malicious clients pretend to be honest, local updates have a chance to be revealed to the adversary. The following theorem shows that these updates do not leak any individual data due to differential privacy.

Theorem E.4. *No adversary in corrupted client set $\mathcal{A}^c \subset C$, where $|\mathcal{A}^c| \leq n-1$, can retrieve the individual values of honest clients.*

Proof. Since we apply differential privacy [20], the local updates cannot leak information regarding the inputs. According to Def. C.1, the added differentially private noise guarantees that the aggregation is indistinguishable whether an individual update participates or not. Therefore, it guarantees the security of individual local updates while aggregation can be calculated. □

F CONVERGENCE ANALYSIS

Let M be the total number of clients in a semi-honest majority client cluster. Semi-honest clients and malicious clients are indexed by $\{1, \cdots, h\}$ and $\{h+1, \cdots, h+m\}$, respectively, where $M = h+m$ and $h > m$ if TNR is greater than 50%. The component j of stochastic gradient and of true gradient are denoted as $\{\tilde{g}_{i,j}\}_{i=1}^{M}$ and g_j respectively. An error probability is shown as follows.

Lemma F.1 (The bound of error probability with malicious clients). *If the TNR (h/M) of the clustering is relatively high, then we have the error probability $\mathbb{P}\left[\mathsf{Sign}\left[\sum_{i=1}^{M}\mathsf{Sign}(\tilde{g}_{i,j})\right] \neq \mathsf{Sign}(g_j)\right] \leq \mathbb{P}_h \cdot O(\sqrt{M/h})$, where $\mathbb{P}_h$ is the bound for the error probability without malicious clients.*

Proof. Every client is a Bernoulli trial with success probability p_h for semi-honest clients and p_m for malicious clients, respectively, to receive the true gradient signs. Let Z_h be the number of semi-honest clients with true signs, which therefore equals the sum of h independent Bernoulli trials, so we know Z_h follows the binomial distribution $B(h, p_h)$. Similarly, we know the number of malicious clients with correct signs Z_m follows the binomial distribution $B(m, p_m)$. Denote $q_h = 1 - p_h$ and $q_m = 1 - p_m$.

Let Z be the total number of clients with true gradient signs, so $Z = Z_h + Z_m$. We use the Gaussian distribution to simplify the analysis. Notice that $B(h, p_h) \sim N(hp_h, hp_hq_h)$ and $B(m, p_m) \sim N(mp_m, mp_mq_m)$, so we get $Z \sim N\left(hp_h + mp_m, hp_hq_h + mp_mq_m\right)$. The event $\mathsf{Sign}\left[\sum_{i=1}^{M}\mathsf{Sign}(\tilde{g}_{i,j})\right] \neq \mathsf{Sign}(g_j)$ is equivalent to event $Z \leq M/2$. Then the error probability equals $\mathbb{P}(Z \leq M/2)$. By using

Cantelli's inequality, we know

$$\begin{aligned}
&\mathbb{P}\left[Z \le M/2\right] = \mathbb{P}[Z \ge 2(hp_h + mp_m) - M/2] \\
&= \mathbb{P}[Z - (hp_h + mp_m) \ge (hp_h + mp_m) - M/2] \\
&\le \frac{1}{1 + \frac{[(hp_h+mp_m)-M/2]^2}{hp_hq_h+mp_mq_m}} \le \frac{\sqrt{hp_hq_h + mp_mq_m}}{2|(hp_h + mp_m) - M/2|} \\
&= \frac{\sqrt{Mp_hq_h}}{2M(p_h - 1/2)} \cdot \frac{\sqrt{h/M + m/M \cdot p_mq_m/p_hq_h}}{(h/M \cdot p_h + m/M \cdot p_m - 1)/(p_h - 1/2)} \\
&= \mathbb{P}_h \cdot O(\sqrt{M/h})
\end{aligned} \tag{1}$$

where the second inequality holds since $\frac{1}{x^2+1} \le \frac{1}{2x}$ for $x > 0$, and the last two equalities hold since $p_h > 1/2$ by [6] and we assume $hp_h > M/2$ with overwhelming probability for a sufficient large TNR. The assumption is reasonable because $hp_h = Mp_h > M/2$ if $TNR = 100\%$. The first factor in Eq. (1) is the bound for the error probability without malicious clients, so we get the error probability less than a $O(\sqrt{M/h})$ factor of that in the case of without malicious clients. □

Let $\boldsymbol{L}$ and $\boldsymbol{\sigma}$ be non-negative losses and standard deviation of stochastic gradients $\tilde{g}$ respectively. $\forall x$, the objective values $f(x)$ are bounded by constants f_* (i.e. $f(x) \ge f_*$). The objective value of 0-th round is referred to as f_0. Under the above conditions, the results are the following.

Theorem F.2 (Non-convex convergence rate of MUDGUARD). *If the TNR of the clustering is relatively high, then the global model generated in the semi-honest cluster converges at a rate*

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=0}^{T-1} \|g_t\|_1\right]^2 \le \frac{1}{\sqrt{N}}\left[\sqrt{\|\mathbf{L}\|_1}\left(f_0 - f_* + \frac{1}{2}\right) + \frac{2}{O(\sqrt{h})}\|\boldsymbol{\sigma}\|_1\right]^2,$$

where N is the cumulative number of stochastic gradient calls up to round T (i.e., $N = O(T^2)$). Therefore, the higher the rate is, the closer the convergence speed is to the case without malicious clients.

Proof. Following the results of Theorem 2 in [6], in the distributed SignSGD with a majority vote, we can get the non-convex convergence rate without malicious clients at

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=0}^{T-1} \|g_t\|_1\right]^2 \le \frac{1}{\sqrt{N}}\left[\sqrt{\|\mathbf{L}\|_1}\left(f_0 - f_* + \frac{1}{2}\right) + \frac{2}{\sqrt{M}}\|\boldsymbol{\sigma}\|_1\right]^2 \tag{2}$$

from

$$|g_t|\mathbb{P}\left[\text{Sign}\left[\sum_{i=1}^{M} \text{Sign}(\tilde{g}_{i,j})\right] \ne \text{Sign}(g_j)\right] \le \frac{\sigma_i}{\sqrt{M}}. \tag{3}$$

As Lemma F.1 proved, in the existence of the malicious clients, we get (3)$\le \frac{\sigma_i}{\sqrt{M}} \cdot O(\sqrt{M/h}) = \frac{\sigma_i}{O(\sqrt{h})}$. By plugging the result into (2), we have the convergence rate of MUDGUARD:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{T}\sum_{t=0}^{T-1} \|g_t\|_1\right]^2 &\le \frac{1}{\sqrt{N}}\left[\sqrt{\|\mathbf{L}\|_1}\left(f_0 - f_* + \frac{1}{2}\right) + \frac{2}{\sqrt{M}}\|\boldsymbol{\sigma}\|_1 \cdot O(\sqrt{M/h})\right]^2 \\
&= \frac{1}{\sqrt{N}}\left[\sqrt{\|\mathbf{L}\|_1}\left(f_0 - f_* + \frac{1}{2}\right) + \frac{2}{O(\sqrt{h})}\|\boldsymbol{\sigma}\|_1\right]^2.
\end{aligned} \tag{4}$$

□

G PROOF OF THEOREM 1

Proof. We denote $a = (a_1, \cdots, a_d)$ and $b = (b_1, \cdots, b_d)$ are two vectors uploaded by malicious clients, where np refers to the number of model parameters:

$$Pr(a_i, b_i = \text{sign}) = \begin{cases} \frac{1}{2}, & \text{sign} = +1 \\ \frac{1}{2}, & \text{sign} = -1 \end{cases}, \forall i \in [d].$$

The adjusted cosine similarity can be computed as:

$$\begin{aligned} COS\_similarity &= \frac{a_1 b_1 + \cdots + a_d b_d}{\sqrt{a_1^2 + \cdots + a_d^2} \cdot \sqrt{b_1^2 + \cdots + b_d^2}} \\ &= \frac{a_1 b_1 + \cdots + a_d b_d}{d}. \end{aligned}$$

Since a_i and b_i are relatively independent, we have:

$$Pr(a_i \cdot b_i = \text{sign}) = \begin{cases} \frac{1}{2}, & \text{sign} = +1 \\ \frac{1}{2}, & \text{sign} = -1 \end{cases}, \forall i \in [d].$$

According to the *Law of large numbers*, $E(COS\_similarity) \sim E(a_i b_i) = 0$. This conclusion can be generalized to any two malicious clients, and malicious clients have the same distance as a semi-honest client. Therefore, if we calculate the adjusted cosine similarity vector of two malicious clients, there should be only two elements of difference. The L_2 distance of these two vectors is $\sqrt{2}$. □

H OTHER EXPERIMENTAL SETUP AND RESULTS

H.1 Other Experiment Setup

We implement MUDGUARD in C++ and Python. We use the MP-SPDZ library [33] to implement secure computations and Pytorch framework [51] for training. All the experiments are conducted on a cluster of machines with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and NVIDIA 1080 Ti GPU, with 32GB RAM in a local area network. As for the cryptographic tools, all the parameters are set to a 128-bit security level.

Datasets. We use MNIST and FMNIST datasets for the image classification task.

• **MNIST [38].** It consists of 60,000 training samples and 10,000 testing samples, where each sample is a 28×28 gray-scale image of handwritten digital (0-9).

• **FMNIST [61].** It contains article images from Zalando and has the same size as MNIST, where each image is a 28×28 gray-scale image associated with a label from 10 classes.

• **CIFAR-10 [35].** It offers 50,000 training samples and 10,000 test samples, where each is a 32×32 color image in a label from 10 different objectives, and there are 6,000 images for each class.

Classifiers. We use LeNet and ResNet-18 to perform training and classification of the datasets.

• **LeNet [37].** Containing 6 layers (including 3 convolution layers, 2 pooling layers, and 1 fully connected layer), LeNet aims to train 44,426 parameters for image classification.

• **ResNet-18 [30].** It provides 18 layers with 11 million trainable parameters to train color images. We use a light vision of ResNet with approx. 2.07 million parameters and complete the experiments with the CIFAR-10 dataset.

Attacks		Baseline	GA	LFA	Krum	Trim	AA	BA
ξ	0.5	0.811	0.803	0.772	0.751	0.763	0.793 / 0	0.797 / 0
	0.6	0.783	0.772	0.77	0.761	0.757	0.784 / 0.002	0.801 / 0
	0.7	0.769	0.767	0.747	0.723	0.726	0.776 / 0.003	0.777 / 0.005
	0.8	0.754	0.752	0.731	0.743	0.754	0.771 / 0.001	0.758 / 0.001
	0.9	0.755	0.737	0.73	0.718	0.724	0.753 / 0.002	0.731 / 0.008
n	10	0.846	0.844	0.824	0.829	0.831	0.836 / 0	0.845 / 0
	50	0.834	0.829	0.829	0.829	0.827	0.827 / 0	0.836 / 0
	100	0.783	0.772	0.77	0.761	0.757	0.784 / 0.002	0.801 / 0
	200	0.774	0.77	0.763	0.771	0.766	0.747 / 0.002	0.771 / 0.002
	500	0.61	0.601	0.61	0.599	0.602	0.615 / 0.015	0.602 / 0.003
q	0.1	0.787	0.787	0.772	0.789	0.786	0.783 / 0	0.787 / 0.004
	0.3	0.788	0.777	0.765	0.773	0.784	0.783 / 0	0.782 / 0.002
	0.5	0.783	0.772	0.77	0.761	0.757	0.784 / 0.002	0.801 / 0
	0.7	0.65	0.637	0.657	0.639	0.65	0.642 / 0.008	0.649 / 0.006
	0.9	0.566	0.542	0.545	0.542	0.548	0.546 / 0.001	0.55 / 0.007

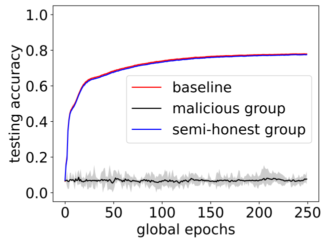
Table 7. Comparison of accuracy with baseline and ASR by an increasing proportion of malicious clients ($\xi \geq 0.5$), #clients n and non-iid degree q, where FMNIST is used. The results under targeted attacks are in the form of "testing accuracy / ASR".

Attacks		Baseline	GA	LFA	Krum	Trim	AA	BA	EA
ξ	0.5	0.573	0.57	0.574	0.557	0.562	0.568 / 0.006	0.572 / 0.007	0.571 / 0.019
	0.6	0.562	0.559	0.559	0.547	0.534	0.521 / 0.011	0.567 / 0	0.568 / 0.03
	0.7	0.54	0.52	0.506	0.513	0.515	0.531 / 0.011	0.524 / 0	0.531 / 0.031
	0.8	0.519	0.508	0.489	0.494	0.488	0.482 / 0.003	0.52 / 0.004	0.501 / 0.05
	0.9	0.489	0.492	0.474	0.45	0.483	0.475 / 0.006	0.484 / 0.008	0.478 / 0.059
n	10	0.677	0.672	0.668	0.665	0.668	0.658 / 0	0.659 / 0.001	0.66 / 0.037
	50	0.641	0.637	0.635	0.653	0.634	0.639 / 0	0.647 / 0.001	0.641 / 0.068
	100	0.562	0.559	0.559	0.547	0.534	0.521 / 0.011	0.567 / 0	0.568 / 0.03
	200	0.46	0.453	0.474	0.457	0.45	0.468 / 0.003	0.446 / 0	0.458 / 0.028
	500	0.27	0.26	0.254	0.274	0.252	0.276 / 0.004	0.262 / 0	0.242 / 0
q	0.1	0.573	0.574	0.566	0.562	0.554	0.555 / 0	0.572 / 0.001	0.558 / 0.058
	0.3	0.567	0.567	0.556	0.535	0.553	0.561 / 0	0.569 / 0	0.543 / 0.064
	0.5	0.562	0.559	0.559	0.547	0.534	0.521 / 0.011	0.567 / 0	0.568 / 0.03
	0.7	0.426	0.417	0.394	0.435	0.424	0.44 / 0.013	0.41 / 0.004	0.429 / 0.015
	0.9	0.229	0.227	0.216	0.224	0.229	0.219 / 0.024	0.217 / 0.013	0.214 / 0.018

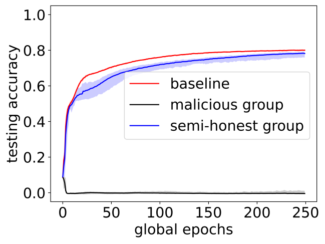
Table 8. Comparison of accuracy with baseline and ASR by an increasing proportion of malicious clients ($\xi \geq 0.5$), #clients n and non-iid degree q, where CIFAR-10 is used. The results under targeted attacks are in the form of "testing accuracy / ASR".

H.2 Other Experimental Results

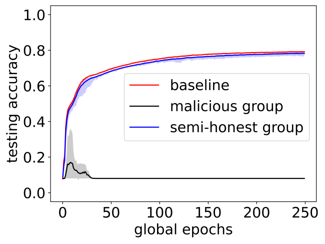
In Figures 5 and 6, we present the comparison of testing accuracy among baseline, semi-honest, and malicious groups under targeted attacks and ASR between the groups under untargeted attacks. In addition, we also give the experimental results by varying proportion of malicious clients ($\xi \geq 0.5$), #clients n and non-iid degree q in Tables 7 and 8. We see that the results are consistent with the analysis in Section 5.1: the untargeted attacks nearly have no impact on the accuracy of the final model (only slightly decreasing the speed of convergence). Since GA may directly upload noise, it can be easily detected from the beginning of the training to the end, resulting in the same convergence as the baseline. For LFA, Krum, Trim, and AA Attacks, MUDGUARD is also difficult



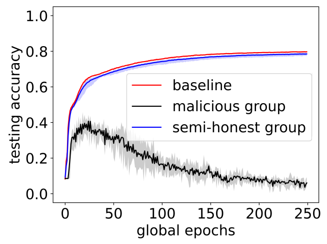
(a) Gaussian Attack



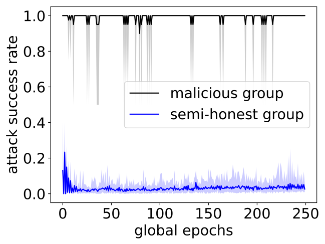
(b) Label Flipping Attack



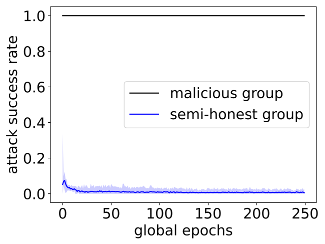
(c) Krum Attack



(d) Trim Attack



(e) Adaptive Attack



(f) Backdoor Attack

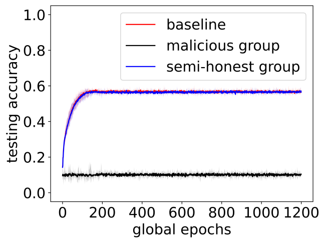
Fig. 5. Comparison of testing accuracy among baseline, semi-honest, and malicious groups under targeted attacks (a-e) and ASR between the groups under untargeted attack (f), where we use LeNet to train FMNIST by default settings in Table 2.

to distinguish between semi-honest and malicious clients at the beginning. Thus, the speed of convergence is slightly decreased. The main difference is that LeNet achieves around 78% in FMNIST, while ResNet-18 provides approx. 56% accuracy in CIFAR-10.

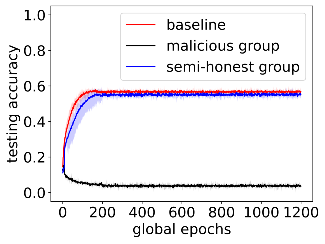
We also provide comparisons with the state-of-the-art Byzantine-robust methods in Figure 7 and 8 on FMNIST and CIFAR-10, respectively. Similar to Figure 4, under the malicious majority of untargeted attacks, the testing accuracies of FLTrust, MUDGUARD, and weights-MUDGUARD are maintained at the same level of the baseline. Under the targeted attacks, FLTrust, MUDGUARD, and weights-MUDGUARD can restrain ASR to about 0%-10%. For more detailed explanations, please refer to Section 5.1.

Impact of α on robustness of MUDGUARD. Turning back to Section 4.2 and Theorem 4.1, the Byzantine-robustness of MUDGUARD relies on whether we can choose a desirable density α. From the experimental results in Section 5.1, we know that EA has a stronger stealthiness than other attacks. Therefore, we use EA as an example in this section to analyze the influence of α on MUDGUARD. Figure 9 shows how the ASR and clustering accuracy varies in semi-honest and malicious groups under EA when we change α. Consistent with Theorem 4.1, once α is larger than $\sqrt{2}$, the malicious and semi-honest clients will cluster together, resulting in MUDGUARD loss of the effectiveness of Byzantine-robustness. A similar situation can be found when α is too small (in this case, all clients will be identified as noise.). From the experiment, we found that 1 is a best practice for α, so we set it as the default parameter. Nevertheless, MUDGUARD cannot 100% guarantee to exclude EA because EA has a strong stealthiness. We will leave this as future work.

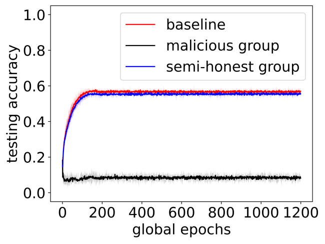
Experimental results on Shakespeare dataset. We further verify the performance of MUDGUARD by Transformer [58] trained on Shakespeare dataset [12] for the next character prediction task under GA, Krum and Trim attacks. We adopt the same default settings for Transformer as in [58]. The rest of the settings of FL follow in Table 2. Table 9 shows that MUDGUARD can capture the



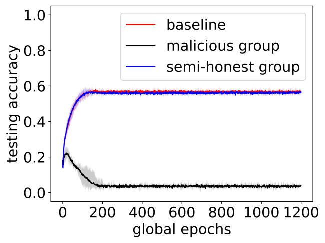
(a) Gaussian Attack



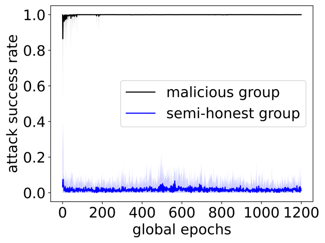
(b) Label Flipping Attack



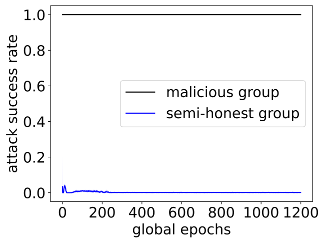
(c) Krum Attack



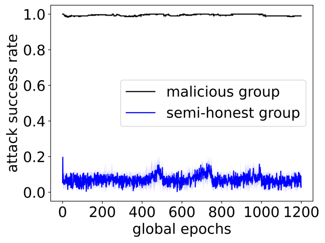
(d) Trim Attack



(e) Adaptive Attack



(f) Backdoor Attack



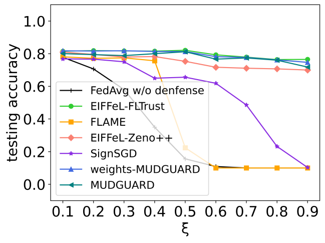
(g) Edge-case Attack

Fig. 6. Comparison of testing accuracy among baseline, semi-honest, and malicious groups under targeted attacks (a-e) and ASR between the groups under untargeted attacks (f-g), where we use ResNet-18 to train CIFAR-10 by the default settings in Table 2.

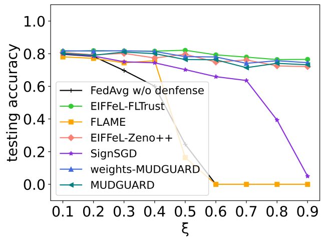
ξ		0.5	0.6	0.7	0.8	0.9
baseline		0.467±0.004	0.459±0.004	0.453±0.005	0.442±0.001	0.428±0.005
Attacks	GA	0.467±0.002	0.458±0.005	0.448±0.002	0.44±0.004	0.422±0.005
	Krum	0.466±0.003	0.46±0.01	0.446±0.006	0.443±0.003	0.427±0.004
	Trim	0.467±0.005	0.459±0.006	0.451±0.003	0.439±0.003	0.432±0.005

Table 9. Comparison of the testing accuracy (mean±std) of baseline and MUDGUARD under GA, Krum, and Trim attacks by increasing the proportion of malicious clients ($\xi \geq 0.5$), where the Shakespeare dataset is used to train the Transformer model.

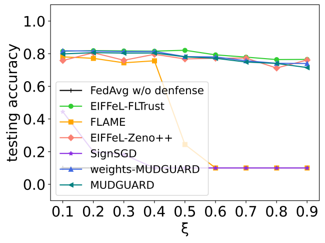
Byzantine-robust features (i.e., under attacks, the testing accuracy can still maintain the same level as the baseline) of the models trained on text datasets. To train these datasets, we use weights or gradients to update the models (in the context of FL). If the models or datasets are poisoned (by malicious clients), malicious updates have differences in updates directions from benign ones. Then MUDGUARD can protect benign clients.



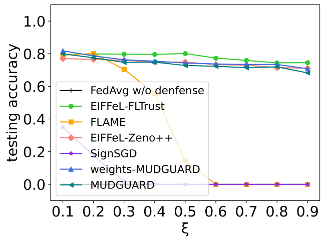
(a) Gaussian Attack



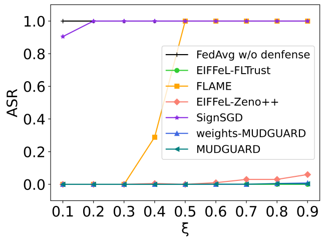
(b) Label Flipping Attack



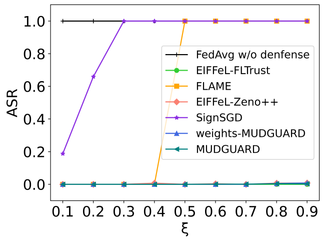
(c) Krum Attack



(d) Trim Attack



(e) Adaptive Attack



(f) Backdoor Attack

Fig. 7. Comparison with Byzantine-robust methods in FMNIST by $\xi = 0.1 - 0.9$.

ξ		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
BA	Lockdown	0.646 / 0.006	0.633 / 0.113	0.62 / 0.145	0.514 / 0.372	0.467 / 0.67	0.224 / 0.963	0.129 / 0.996	0.101 / 1	0.1 / 1
	MUDGUARD	0.648 / 0.002	0.645 / 0.003	0.623 / 0.007	0.614 / 0.005	0.572 / 0.007	0.567 / 0	0.524 / 0	0.52 / 0.004	0.484 / 0.008
LFA	GAS	0.644	0.625	0.634	0.607	0.598	0.542	0.54	0.528	0.505
	MUDGUARD	0.646	0.634	0.63	0.613	0.593	0.572	0.549	0.545	0.534

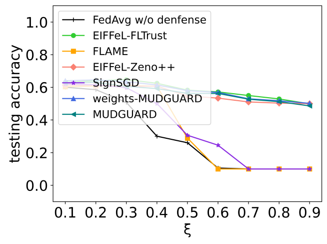
Table 10. Comparison with other Byzantine-robust methods in CIFAR-10 by ξ=0.1-0.9. The results under targeted attacks are in the form of “testing accuracy / ASR"

Robustness comparison against other methods. We compare MUDGUARD with two recent works on Byzantine-robust FL (i.e., Lockdown [31] and GAS [41]) in CIFAR-10. Lockdown [31] presents a defense against backdoor attacks in FL by employing isolated subspace training. This approach involves partitioning the model training process into distinct subspaces for different clients, incorporating randomness in the training process, and utilizing quorum consensus. GAS [41] addresses the challenges of Byzantine robustness in FL under non-iid settings. It involves splitting high-dimensional gradients into low-dimensional sub-vectors, applying robust aggregation rules to these sub-vectors to identify honest gradients, and then aggregating the identified honest gradients. Since Lockdown and GAS are defenses against targeted and untargeted attacks, we compare them with MUDGUARD under BA and LFA, respectively. As shown in Table 10, MUDGUARD outperforms these defenses because they operate under the assumption of an honest majority.

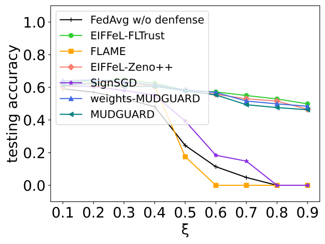
I DISCUSSION

I.1 Defending Against Other Attacks

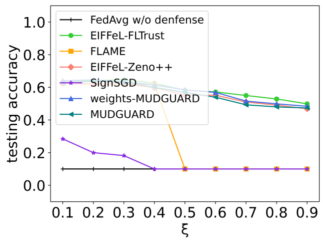
In the experiments, we consider SOTA (un)targeted attacks. We say that interested readers may use other attacks to test MUDGUARD, in which Byzantine-robustness could not be seriously affected. We take the Distributed Backdoor Attack (DBA) [62] as an example. DBA decomposes a global trigger into several pieces distributed to local clients. It, however, yields significant changes to



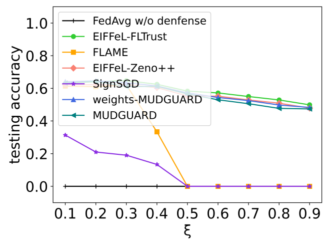
(a) Gaussian Attack



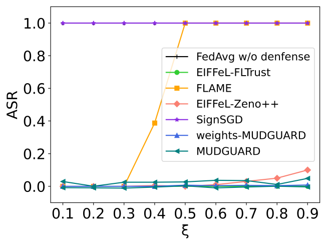
(b) Label Flipping Attack



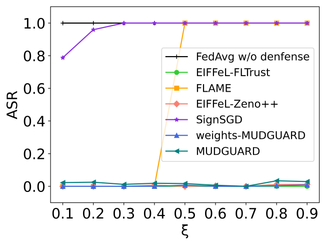
(c) Krum Attack



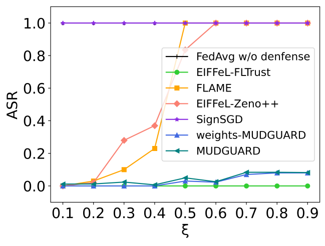
(d) Trim Attack



(e) Adaptive Attack

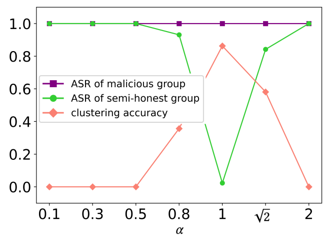


(f) Backdoor Attack

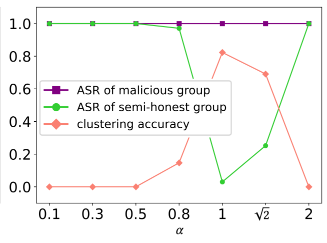


(g) Edge-case Attack

Fig. 8. Comparison with Byzantine-robust methods in CIFAR-10 by $\xi = 0.1 - 0.9$.



(a) MNIST



(b) CIFAR-10

Fig. 9. Impact of α on the robustness of MUDGUARD, where EA and default settings are used.

some dimensions of updates to maintain the ASR of the backdoor task. Since the cosine distance between malicious and benign updates is distinguishable, MUDGUARD can still work well under DBA. Note another attack, Little Is Enough [4], have not been considered in this work because it requires attackers to have knowledge of the gradients of semi-honest clients, which violates privacy preservation.

I.2 Advantages of Adjusted Cosine Similarity

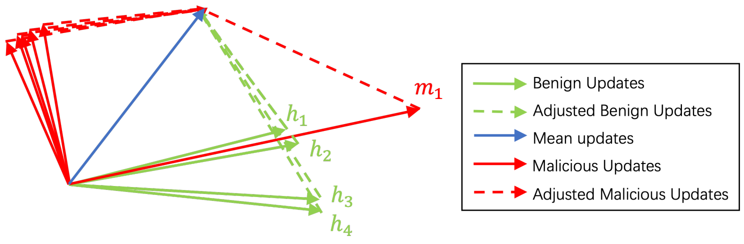


Fig. 10. An example of calculation of adjusted cosine similarity

As shown in Figure 10, we present an example of the calculation of adjusted cosine similarity. It is clear to see that adjusted cosine similarity can capture the magnitudes and directions of updates by transferring updates to adjusted updates. Although m_1 does not have too many differences in directions (i.e., m_1 will be clustered with h_1 and h_2), its differences with h_1 and h_2 in magnitudes can be easily captured by CosM. Furthermore, due to non-iid, the (h_1, h_2) and (h_3, h_4) will be clustered into two groups if cosine distance is applied. However, by subtracting mean updates, this influence can be reduced.

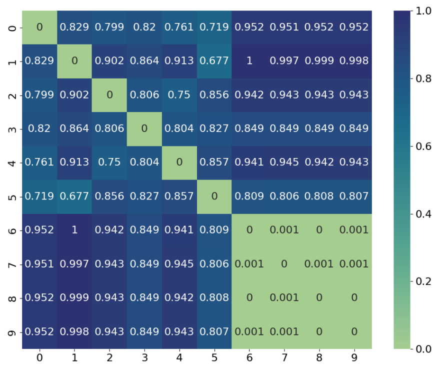
We also show the experimental results in Figure 11, where FMNIST is used under BA, and the number of clients is set to 10 (the first six are benign clients, the rest are malicious clients). The rest of the default settings follow Table 2. From Figure 10, we can see that if only cosine distance is calculated, honest updates will be classified as noise due to the influence of non-iid. The adjusted cosine similarity weakens this effect (Figure 10b). Calculating the L_2 distance again will make the distinction between the two groups more obvious (Figure 10c).

Furthermore, we provide a comparison when MUDGUARD uses cosine similarity and adjusted cosine similarity for clustering in Figure 12. It is clear to see that the testing accuracy of MUDGUARD with cosine similarity abruptly goes down when the model approaches convergence. On the contrary, this does not happen in MUDGUARD with adjusted cosine similarity. The detailed explanation is given in Section 4.2.

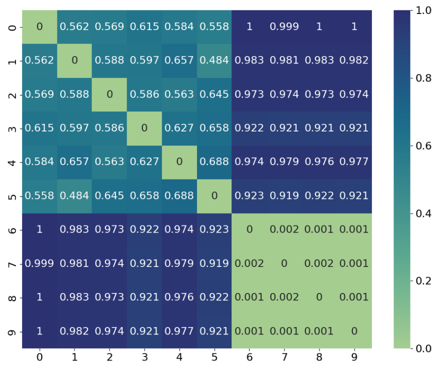
I.3 Dynamic Attacks

Recall that MUDGUARD can perform well in terms of testing accuracy under the assumption that malicious clients consistently perform one type of attack (e.g., GA) throughout the whole training period. It can also perform well if we allow malicious clients to perform different attacks on the epochs, e.g., GA to the first 10 epochs and then Krum attacks to the remaining epochs. We notice that all the attacks (we consider in this work) except GA may require several epochs of training (as a buffer) to produce attack effects. But these buffer epochs can boost MUDGUARD's clustering performance. This is so because the clustering ability is enhanced with the increase of training rounds. On the other hand, the TNR and TPR (of the clustering) could be relatively low in these epochs. Some malicious clients can be clustered into a semi-honest group, but this will not seriously affect the accuracy performance of the model.

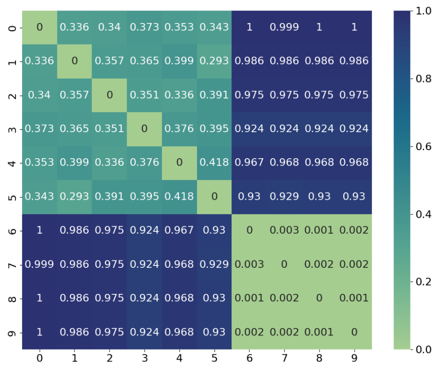
One may think malicious clients are allowed to perform all the attacks in a single epoch. But so far, it is unknown how to group those attacks together friendly and meanwhile maximize their attack effects. In practice, the attacks may deliver an update in different directions, and further, they may even yield influence on each other. For example, GA could easily destroy the convergence of BA. We say that it is an interesting open problem to consider launching GA, LFA, Krum, Trim, and AA attacks together in an epoch to evaluate the accuracy and ARS.



(a) Pairwise cosine distance



(b) Pairwise adjusted cosine distance (CosM)



(c) Pairwise L_2 distance for CosM

Fig. 11. Calculation results of pairwise distance.

I.4 Varying Clients Subsampling Rate

We assert that MUDGUARD can perform well under different clients' subsampling rates. This benefits from the proposed *Model Segmentation* that can resist malicious-majority clients. Imagine that in the context of an honest majority (e.g., 40 out of 100 are malicious), if the subsampling rate is set

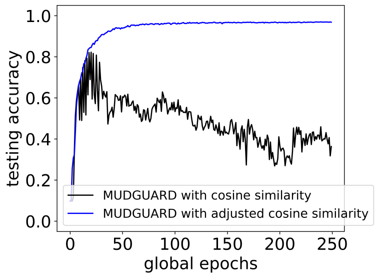


Fig. 12. Comparison of MUDGUARD with cosine similarity and adjusted cosine similarity under GA.

relatively low (e.g., 10%), we eventually have a malicious majority case in one training epoch with a high probability. Our experimental results have demonstrated that MUDGUARD can still achieve Byzantine-robustness under a low subsampling rate.

I.5 Learning Rate and Local Epoch

They are subtle parameters that decide the performance of FL training. A low learning rate can slow down the speed of convergence, and on the other hand, a high rate hinders the model's convergence, harming accuracy. As for the local epoch, in the case of iid, if carefully increasing the number of epochs, we can make the model converge fast. But under a large degree of non-iid (e.g., q=0.5), the increase of epoch leads to updates in different directions, making the FedAvg algorithm invalid [43]. In this work, we set these two parameters according to the recommendations given in [6, 43]. Exploring their impacts on training is orthogonal to the main focus of this work.

I.6 Privacy-preserving DBSCAN

This is one of the core parts we used to build MUDGUARD. It can apply to other real-world domains, e.g., anomaly detection and encrypted traffic analytics, where data should be clustered securely. But we note that the current MUDGUARD with optimization may not scale well in these applications. We did the optimization for the sake of efficiency by using SignSGD and binary secret sharing, which cannot support precise floating-point arithmetic.

I.7 Limitations

Using weights as updates. To provide cost-effective secure computations, the proposed MUDGUARD only implements the update method by SignSGD. If we use weights as updates, the secret shares sent to the servers will be in floating-point format. In this case, we will have to downgrade the design to the unoptimized MUDGUARD in Table 5, which could cause a considerable amount of both communication costs and runtime. An interesting future work could be to propose a more lightweight (than the current design) and secure MPC framework for MUDGUARD.

Performance under EA. Although MUDGUARD does achieve good performance in terms of accuracy and (to some extent) efficiency, under the EA, MUDGUARD's ASR cannot be eventually reduced to nearly 0%. In future work, we will improve the TNR and TPR of the clustering algorithm so as to recognize subtle differences between malicious and semi-honest clients.

Received August 2024; revised September 2024; accepted October 2024