Efficient, Private and Robust Federated Learning

Meng Hao
University of Electronic Science and Technology of China
menghao@std.uestc.edu.cn

Hongwei Li*
University of Electronic Science and Technology of China
hongweili@uestc.edu.cn

Guowen Xu
Nanyang Technological University
guowen.xu@ntu.edu.sg

Hanxiao Chen
University of Electronic Science and Technology of China
hanxiaochen@std.uestc.edu.cn

Tianwei Zhang
Nanyang Technological University
tianwei.zhang@ntu.edu.sg

ABSTRACT

Federated learning (FL) has demonstrated tremendous success in various mission-critical large-scale scenarios. However, such promising distributed learning paradigm is still vulnerable to privacy inference and byzantine attacks. The former aims to infer the privacy of target participants involved in training, while the latter focuses on destroying the integrity of the constructed model. To mitigate the above two issues, a few works recently explored unified solutions by utilizing generic secure computation techniques and common byzantine-robust aggregation rules, but there are two major limitations: 1) they suffer from impracticality due to efficiency bottlenecks, and 2) they are still vulnerable to various types of attacks because of model incomprehensiveness.

To approach the above problems, in this paper, we present SecureFL, an efficient, private and byzantine-robust FL framework. SecureFL follows the state-of-the-art byzantine-robust FL method (FLTrust NDSS'21), which performs comprehensive byzantine defense by normalizing the updates' magnitude and measuring directional similarity, adapting it to the privacy-preserving context. More importantly, we carefully customize a series of cryptographic components. First, we design a crypto-friendly *validity checking* protocol that functionally replaces the normalization operation in FLTrust, and further devise tailored cryptographic protocols on top of it. Benefiting from the above optimizations, the communication and computation costs are reduced by half without sacrificing the robustness and privacy protection. Second, we develop a novel *preprocessing* technique for costly matrix multiplication. With this technique, the directional similarity measurement can be evaluated securely with negligible computation overhead and zero communication cost. Extensive evaluations conducted on three real-world datasets and various neural network architectures demonstrate that SecureFL outperforms prior art up to two orders of magnitude in efficiency with state-of-the-art byzantine robustness.

*corresponding author

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488014

KEYWORDS

Federated learning, Privacy protection, Byzantine robustness.

ACM Reference Format:
Meng Hao, Hongwei Li, Guowen Xu, Hanxiao Chen, and Tianwei Zhang. 2021. Efficient, Private and Robust Federated Learning. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3485832.3488014

1 INTRODUCTION

Federated learning (FL) [33], as a promising distributed learning paradigm, has shown its potential to facilitate real-world applications like Gboard mobile keyboard [39] [47], electronic health records mining [19] [31] and credit risk prediction [28]. Roughly speaking, in FL multiple participants (e.g. mobile devices) collaboratively train a global model via exchanging local updates (i.e., gradients) under the orchestration of a service provider, while keeping the training data decentralized. Despite such advantages, existing FL approaches still suffer from privacy inference [52] [21] [20] and byzantine attacks [4] [8] [5]. In the former, adversaries can infer private information (e.g., sensitive training data) of target parties from the local updates. Particularly in medical diagnosis applications, patients' private information such as medical conditions may be leaked from an unprotected FL system [37]. It fundamentally violates current strict regulations, such as General Data Protection Regulation (GDPR). The latter focuses on corrupting the global model's accuracy and convergence by submitting elaborate poisoning updates, which will cause serious security threats. For instance, once the underlying FL models of autonomous driving suffer such attacks, it will cause misclassification and hence severe traffic accidents [4].

To mitigate the information leakage issue, a variety of secure aggregation schemes based on cryptographic protocols [2] [50] [7] [10] have been proposed and employed in improving the original FL systems. For example, several works use homomorphic encryption (HE) [2] [50] to encrypt the parties' gradients. After that, the service provider can aggregate such encrypted gradients without decryption due to the homomorphic nature of HE. Besides, secure multi-party computation (MPC) enables parties to jointly perform arbitrary function evaluations over their inputs while keeping those inputs private, which is also used in secure aggregation in FL systems [7] [10] [46]. Consequently, the local gradients of parties are obfuscated and only the aggregated update is revealed. On the other hand, many works have made rapid strides towards

Table 1: Comparison with prior works on properties necessary for federated learning

Proposed Works	Approach	Privacy	Robustness	Scalability	User Efficiency
[10]	MPC+FedAvg	●	○	○	○
[7]	MPC+FedAvg	●	○	●	○
[2]	HE+FedAvg	●	○	○	○
[22]	HE+FedAvg	●	○	○	○
[25]	MPC+Krum	◐	◐	○	●
[44]	MPC+Krum	◐	◐	○	○
[30]	MPC+Cosine Similarity	◐	◐	○	●
Our SecureFL	**Crypto-friendly FLTrust with customized MPC&HE**	●	●	●	●

realizing byzantine robustness in FL via developing statistically robust aggregation rules, such as Krum [9], Median [49], and Bulyan [16]. The main idea is that the service provider removes suspicious outliers by performing statistical analysis among parties' local gradients, before updating the global model with them (see Section 2 for more details). Notably, Cao et al. recently proposed the state-of-the-art byzantine robust FL method, FLTrust [11], which performs effective and comprehensive byzantine detection by introducing a novel server update as the baseline and analyzing both the direction and magnitude of local gradients. Specifically, they first designed a normalization protocol to prevent the malicious parties' manipulation on magnitude, and then performed directional similarity measurement to eliminate the effect of local gradients that are in the opposite direction from the server update.

Although many works have been proposed to alleviate the problems of privacy leakage and byzantine attacks, most of them unilaterally separate the above two concerns and underestimate their internal connections. Essentially, privacy violation and byzantine attacks are intricately intertwined. Attackers may carefully exploit byzantine vulnerabilities to infer other parties' training datasets and hence destroy even privacy-protected FL systems [25], while privacy leakage provides adversaries with more favorable prior knowledge to launch omniscient and adaptive byzantine attacks [18] [8]. Therefore, it is necessary to design a FL system realizing privacy protection and byzantine robustness simultaneously. To this end, a natural approach is to integrate generic cryptographic techniques such as MPC and HE [14] [17] with existing byzantine-robust FL protocols [9] [11]. However, it adds a large overhead due to the evaluation of heavy cryptographic operations, e.g. large-scale matrix multiplication in the measurement of the parties' gradient quality, and complex non-linear function used to privately exclude outlier gradients. Consequently, the challenging problem is how to design customized cryptographic protocols for private FL systems that can efficiently implement byzantine defense at the same time. A few works [43] [24] [29] explored to develop unified solutions (see Section 2 for more details), however, to "purely" facilitate the design of efficient cryptographic protocols, they made inappropriate trade-offs, such as revealing intermediate values (e.g., the quality of the parties' updates) or exploiting simple but vulnerable aggregation rules [9].

In this paper, we introduce SecureFL, an efficient, private and byzantine-robust FL framework that approaches the above problems, as illustrated in Table 1. Our SecureFL follows the state-of-the-art byzantine robust method, FLTrust [11], and adapts it to the privacy-preserving context to achieve full privacy protection. We mainly focus on reducing the overhead of evaluating byzantine detection under ciphertext. Specifically, inspired by the respective advantages of HE and MPC, we devise customized cryptographic protocols for the two key steps of FLTrust. **(1) For the magnitude normalization, the key idea is to design a crypto-friendly alternative that functionally replaces the costly normalization operation.** Specifically, we observe that this step involving d (i.e., the size of the gradient) reciprocal square root and one high-dimensional inner product is computationally expensive in secure computation. To reduce this overhead, we leave the implementation of normalization to the party side in plaintext, and design a crypto-friendly *validity checking* protocol for the service provider to inspect whether parties deviate from the specification and explicitly exclude updates with wrong form. Along with our customized MPC protocols, our solution reduces the communication and computation costs roughly by half, without any privacy leakage and robustness loss. **(2) For the directional similarity measurement, our main insight is that the service provider can pre-compute some cryptographic protocols before the parties' local gradients are available.** Specifically, we identify and repurpose an important but under-utilized phase, called *preamble phase*, where in prior works the service provider only stays idle and waits for the parties to upload their local gradients. In our SecureFL, utilizing advanced HE techniques that perform well in linear function evaluations, we develop a novel preprocessing protocol for matrix multiplication operations (i.e., the center building block of this step), which is evaluated in the preamble phase to accelerate the privately robust aggregation procedure. Beneficially, once the parties' local gradients are available, the secure directional similarity measurement can be implemented non-interactively with negligible computation overhead. Notably, our cryptographic recipes may be of independent interests, and can be used in other byzantine-robust FL schemes [9] [16] and even a wider range of scenarios such as scientific computing [13]. Besides, to demonstrate the efficiency and robustness of SecureFL, we conduct extensive experiments on three real-world datasets with various neural network architectures against the latest two types of byzantine attacks.

Our key contributions can be summarized as follows:

- We propose a new federated learning framework, SecureFL, which achieves state-of-the-art robustness, full privacy protection and efficiency at the same time.
- We carefully customize a series of cryptographic components for enjoying efficiently mathematical operations in the evaluation of privately robust aggregation.
- Extensive experiments illustrate that SecureFL outperforms prior art up to two orders of magnitude in efficiency with state-of-the-art robustness.

The remainder of this paper is organized as follows. In Section 2, we overview the latest related works. In Section 3, we introduce the system model and the threat model followed by describing the design goals and cryptographic primitives. Then, we present our cypto-friendly FL variant in Section 4 and carry out our SecureFL framework in detail in Section 5. Finally, the performance evaluation is discussed in Section 6 and we conclude our work in Section 7.

2 RELATED WORKS

In this section, we overview the latest related works about privacy-preserving and byzantine-robust FL.

McMahan et al. [33] developed the pioneering FL method in non-adversarial settings, FedAvg, which computes the average of the local model updates, but a risk (i.e., the correctness of the learned model) emerges upon facing even a single byzantine party [9]. To mitigate such problem, Blanchard et al. proposed Krum [9], which selects as the global update one of the n local gradients that is similar to other parties' gradients based on the Euclidean distance. After that, Bulyan [16] was designed via combining Krum and the idea of median to further improve the byzantine robustness. Besides, Yin et al. [49] proposed coordinate-wise aggregation rules, such as trimmed mean and median based rules. Specifically, for each i-th coordinate of the gradient vector, the service provider first sorts the n local gradients and then takes their median (or trimmed mean) as the i-th parameter of the global update. While the above methods were claimed to be robust against byzantine failures at the time, the latest works [18] [5] [45] [41] found that they are still vulnerable to sophisticated byzantine attacks. Consequently, Cao et al. proposed FLTrust, the state-of-the-art byzantine-robust method that achieves robustness against even strong and adaptive attacks [18]. Nevertheless, privacy issues remain unresolved in the aforementioned byzantine-robust works.

Recently, several works [24] [43] [22] [29] [38] began to explore unified solutions to privacy leakage and byzantine attacks. For example, He et al. [24] combined additive secret sharing based secure computation and a variant of the Krum aggregation protocol [9], to mitigate the above problems. Concurrently, So et al. [43] also devised a similar scheme based on the Krum aggregation rule, but relied on different cryptographic techniques including verifiable Shamir's secret sharing and Reed-Solomon code. To further improve byzantine robustness, [29] proposed a new defense by using cosine similarity and generic secure multi-party computation tools. Different from the Krum-based works [24] [43] that utilize the Euclidean distance, the cosine similarity measurement additionally takes into account the direction of parties' updates. However, unlike SecureFL, the above approaches incur a significant communication and computation cost that grows quadratically with the number of parties. The main reason is that they perform pair-wise byzantine statistical analysis between parties' gradients, such that each gradient is required to be compared with all other gradients. Another disadvantage is that they leak intermediate information such as confidence information that is used to measure the quality of parties' gradients. Such seemingly inconspicuous contents may reveal the parties' identity or data quality, thereby undermining fairness. Besides, Nguyen et al. [38] proposed FLGuard to realize full privacy protection but focused on mitigating backdoor attacks [4]. To this end, they design a clustering-based cosine similarity measurement and combine existing secure computation techniques. However, its reliance on clustering computations and heavy cryptographic operations results in a protocol that is more expensive than SecureFL's protocol with respect to both computation and communication. Another recent work [22] for private byzantine-robust FL was designed using Trusted Execution Environments (TEEs). In their solution, the service provider is equipped with a TEE, in which the robustness aggregation procedure is performed using their customized methods without compromising the parties' privacy. However, the latest research [44] shows that TEEs are still vulnerable to hardware-based side-channel attacks, and hence their work may suffer potential security risks. Different from TEE-based solutions, our SecureFL provides formal security proof and robustness analysis theoretically (refer to Appendix B). More importantly, all of the above schemes are still vulnerable to strong byzantine attacks [18] [5], since they make an inappropriate trade-off between the cryptographic performance and robustness, namely that they use simple but vulnerable aggregation rules, such as Krum [9]. In contrast, our SecureFL follows the state-of-the-art byzantine-robust FLTrust [11], and does not sacrifice inference accuracy and robustness even in designing crypto-friendly FL alternatives and customized cryptographic protocols.

3 PRELIMINARIES

In this section, we first introduce the system model and the threat model following the prior works [43] [24], and then describe the design goals and the cryptographic primitives.

3.1 System Model

In our SecureFL, there are a set of parties $P_1, P_2, ..., P_n$ and two servers (i.e., the FL service provider SP and the computing server CS). SP coordinates the whole training process and CS assists SP in performing secure two-party computation (2PC). Assume each party P_i holds a local dataset $\mathcal{D}_i$, $i \in [n]$ and SP holds a small but clean seed dataset $\mathcal{D}_s$ for byzantine detection (see more details in Section 4). We denote the overall training dataset as $\mathcal{D} = \bigcup_{i\in[n]} \mathcal{D}_i$. The goal of n parties is to collaboratively train a global model by solving the following optimization problem: $\min_{\boldsymbol{\omega}} \mathbb{E}_{\mathcal{D}}[\mathcal{L}(\mathcal{D}, \boldsymbol{\omega})]$, where $\boldsymbol{\omega}$ is the weight of the global model and $\mathcal{L}$ is the loss function, e.g., the cross-entropy loss function. In practice, stochastic gradient descent (SGD) is widely used to minimize the aforementioned loss in FL. Figure 1 shows the system model of our SecureFL, and we can observe it consists of three steps in each iteration. Specifically, at step I, SP broadcasts the latest global model to parties that are

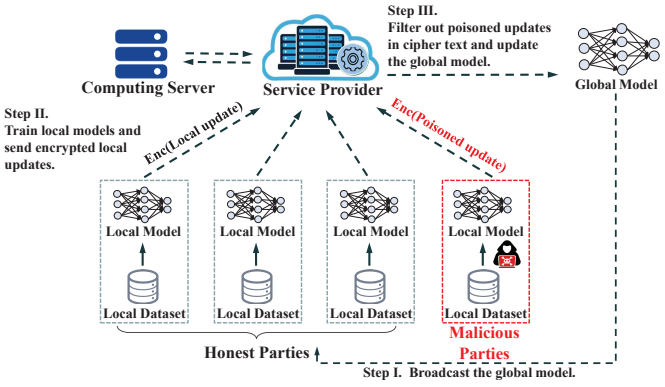


Figure 1: System Model

selected in this iteration. Then at step II, each party p_i locally trains the global model received, computes its local gradient $\boldsymbol{g}_i = SGD(\mathcal{D}_i, b, \boldsymbol{\omega})$ where b is the batch size, and secretly shares $\boldsymbol{g}_i$ with the two servers. Finally at step III, SP and CS privately aggregate the received local gradients by exploiting the robust aggregation scheme, and update the global model.

3.2 Threat Model

In our SecureFL, there are two types of adversaries: malicious parties that aim to actively corrupt the global model by sending poisoning gradients, and honest-but-curious servers (i.e., SP and CS) that follow the privately robust aggregation protocol but try to passively infer information about target party's training data $\mathcal{D}_i$. Typically, malicious parties have the following knowledge: the local training data and local gradients of the corrupted parties, the training algorithm, the loss function, and the local learning rate. The latter honest-but-curious servers have access to all parties' local gradients, the aggregation algorithm and the seed dataset $\mathcal{D}_s$. This setting is reasonable and also consistent with real-world FL systems [38]. Namely, for maintaining a good reputation to provide more FL services, the servers (e.g., Google and Amazon) are unwilling to be caught acting maliciously, but parties may have various bad motives such as for competitive purposes to maliciously corrupt the ongoing FL systems.

Remark 1. We consider two non-colluding servers. This setting is actually weaker than the single-server setting, but has been widely formalized and instantiated in previous works, especially in the machine learning field such as Prio (USENIX NSDI'17) [13], SecureML (IEEE S&P'17) [36] and Quotient (ACM CCS'19) [1]. There are two key advantages in this setting: 1) Parties can significantly reduce local overheads by outsourcing the privately robust aggregation procedure to the two servers. This is in line with the design goal of SecureFL, which is not to incur extra computation and communication costs for parties. 2) We can benefit from a combination of efficient secure 2PC techniques [14] for boolean circuits such as the evaluation of compare [40] and arithmetic circuits such as multiplication procedures [6]. As shown in subsequent sections, our privately robust aggregation protocol, involving matrix multiplication and complex non-linear function, is efficiently evaluated using the customized secure 2PC protocols we designed. Moreover, several studies have been applied to practical scenarios; e.g., Prio [13], a system for the privacy-preserving collection of aggregate statistics, has been adopted by Mozilla's Firefox browser[1] and used in iOS and Android[2] to measure the effectiveness of their Exposure Notification systems. In summary, the setting of two non-colluding servers not only facilitates the design of efficient cryptographic protocols, but also demonstrates potential commercial applications.

3.3 Design Goals

SecureFL aims to empower a FL framework that achieves byzantine robustness against malicious parties, scalability and privacy protection at the same time without sacrificing the inference accuracy and efficiency at the party side. More specifically, we aim to achieve the following goals:

- **Robustness against malicious parties**: Considering real-world business competition, malicious parties may launch byzantine attacks to deliberately destroy the competitor's FL system. Our goal is to preserve the accuracy and robustness of the trained model against such malicious parties.
- **Privacy protection**: In the training process, parties will submit to the service provider their local updates that contain the private information of their training dataset. Therefore, the privacy of parties should be protected from being leaked.
- **Scalability**: The privately robust aggregation protocol should be implemented efficiently, and hence applied to large-scale FL systems that involve hundreds of parties and advanced neural network architectures.
- **Efficient protocols for parties**: FL is particularly used in the setting of resource-constrained mobile devices, in which communication is extremely expensive. Therefore, compared with vanilla FL [33], the proposed scheme should not incur extra computation and communication costs for parties.

3.4 Cryptographic Primitives

In this section, we provide a description of the cryptographic building blocks used in SecureFL.

3.4.1 Packed Linearly Homomorphic Encryption. A packed linearly homomorphic encryption (PLHE) scheme [27] [51] is an encryption scheme, which additionally enables packing multiple messages into a single ciphertext and hence supports SIMD (single instruction multiple data) [42] linearly homomorphic operations. In details, a PLHE scheme is a tuple of algorithms PLHE = (KeyGen; Enc; Dec; Eval) with the following syntax: 1) $\mathsf{KeyGen}(1^k) \rightarrow (pk, sk)$: on input a security parameter k, KeyGen is a randomized algorithm that outputs a public key pk and a secret key sk. 2) $\mathsf{Enc}(pk, \mathbf{m}) \rightarrow \mathbf{c}$: the encryption algorithm Enc takes a packed plaintext message $\mathbf{m}$ and encrypts it using pk into a ciphertext $\mathbf{c}$. 3) $\mathsf{Dec}(sk, \mathbf{c}) \rightarrow \mathbf{m}$: on input sk and a ciphertext $\mathbf{c}$, the (deterministic) decryption algorithm Dec recovers the plaintext message $\mathbf{m}$. 4) $\mathsf{Eval}(pk, \mathbf{c}_1, \mathbf{c}_2, \mathsf{func}) \rightarrow \mathbf{c}$: on input pk, two (or more) ciphertexts $\mathbf{c}_1, \mathbf{c}_2$ containing $\mathbf{m}_1, \mathbf{m}_2$,

[1] https://blog.mozilla.org/security/
[2] https://github.com/google/exposure-notifications-android

and a linear function func, Eval outputs a new ciphertext $\mathbf{c}$ encrypting $\text{func}(\mathbf{m}_1, \mathbf{m}_2)$. Our SecureFL builds on the Brakerski-Fan-Vercauteren (BFV) scheme [17], which is one of the state-of-the-art PLHE solutions.

3.4.2 Secret Sharing. There are two common secret sharing schemes: *additive secret sharing* and *boolean secret sharing*. To additively share x in a ring $\mathbb{Z}_p$, the first party samples $r \in \mathbb{Z}_p$ uniformly at random, and sends $x-r \in \mathbb{Z}_p$ to the second party. In this paper, we denote an additive secret sharing of x as a pair of $(\langle x\rangle_0, \langle x\rangle_1) = (r, x-r) \in \mathbb{Z}_p^2$. In *boolean sharing*, we denote the *boolean-share* of $x \in \mathbb{Z}_2$ as $\langle x\rangle_0^B$ and $\langle x\rangle_1^B$, and the shares satisfy $x = \langle x\rangle_0^B \oplus \langle x\rangle_1^B$. Arithmetic operations can still be executed in the form of sharing using Beaver's multiplicative triples (show details in Appendix C).

3.4.3 Oblivious Transfer. In the 1-out-of-2 oblivious transfer (OT) protocol, one party called sender holds two messages x_0, x_1, and the other party called receiver has a choice bit b. After the OT protocol, the receiver learns x_b without obtaining anything about x_{1-b}, while the sender learns nothing about b. Moreover, a widely used technique is OT extension [26], which implements a large number of OTs using only symmetric cryptographic primitives via a few base OTs. Besides, on top of the OT extension, one important variant is correlated OT (COT) [3]. Particularly, the sender inputs a correlation function $f()$, and obtains a x_0 randomly chosen by the protocol itself and a correlated $x_1 = f(x_0)$. By doing so, the communication bandwidth from the sender to the receiver is reduced by half.

3.4.4 Pseudorandom Generator. A Pseudorandom Generator (PRG) [48] takes as input a uniformly random seed and generates a long pseudorandom string. The security of PRG ensures that the output of the generator is indistinguishable from the uniform distribution in polynomial-time, as long as the seed is hidden from the distinguisher. PRG is instantiated in our SecureFL to cut the communication to half between each party and the servers.

4 CRYPTO-FRIENDLY BYZANTINE-ROBUST FL PROTOCOL

In this section, we first revise the state-of-the-art byzantine-robust FL framework, FLTrust [11], and then we propose an improved crypto-friendly variant.

4.1 Revising FLTrust

The main idea of FLTrust [11] is that SP collects a small but clean seed dataset, and computes a server update $\boldsymbol{g}_s$ on it as the baseline to detect and exclude byzantine parties. Specifically, they first normalizes each local gradient by scaling it to have the same magnitude. Then, SP assigns a trust score to each local gradient, where the trust score is larger if the direction of the local model update is more similar to that of the server update. Formally, it is realized through cosine similarity measurement along with ReLU-based clipping. The training process of FLTrust is consistent with most FL protocols except for the robust aggregation procedure, consisting of the following steps:

- **Normalizing gradients' magnitude.** FLTrust first normalizes each local gradient as follows:
$$\tilde{\boldsymbol{g}}_i = \frac{\boldsymbol{g}_i}{\|\boldsymbol{g}_i\|} \tag{1}$$
where $\|\cdot\|$ denotes the ℓ_2 norm. The role of normalization is to alleviate such manipulation that attackers may scale the magnitude of local gradients by a large factor.
- **Measuring gradients' direction similarity.** FLTrust then assigns a trust score TS_i to each local gradient $\boldsymbol{g}_i$, by computing its consine similarity cos_i with the server gradient $\boldsymbol{g}_s$ and clipping the consine similarity via ReLU[3]. Formally, the trust score is defined as follows:
$$TS_i = \text{ReLU}(cos_i) = \text{ReLU}(\langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_s \rangle) \tag{2}$$
where $\tilde{\boldsymbol{g}}_i$ and $\tilde{\boldsymbol{g}}_s$ are the normalized local and server gradients, respectively. The role of the ReLU function is to exclude the local gradient that have a negative impact (i.e., negative cosine similarity) on the global model update.
- **Aggregating weighted gradients.** FLTrust finally aggregates the normalized local gradients weighted by their trust scores, and scales the aggregation result as follows:
$$\boldsymbol{g}^{global} = \frac{\|\boldsymbol{g}_s\|}{TS} \sum_{i=1}^{n} TS_i \cdot \tilde{\boldsymbol{g}}_i, \tag{3}$$
where $TS = \sum_{i=1}^{n} TS_i$, and the magnitude of the aggregation result is same as $\boldsymbol{g}_s$.

On the basis of FLTrust, it is trivial to design a privacy-preserving protocol by utilizing state-of-the-art mixed-protocol 2PC frameworks such as ABY[14]. Specifically, linear operations (such as matrix multiplication) are computed in the additive sharing, while the computation of reciprocal square root and ReLU can be realized in the boolean sharing. Despite having realized a privately robust aggregation protocol, it still suffers from two key efficiency issues: 1) As described in previous works [35] [40], the normalization in FLTrust is a costly operation in 2PC, since the involved reciprocal square root operations are computationally expensive. 2) The directional similarity measurement for all parties can be formalized as a matrix-vector multiplication, which is time-consuming due to the high dimension of model gradients and the large number of parties. In Section 6, we compare against such baseline scheme.

4.2 Crypto-friendly byzantine-robust FL protocol

In the following, we first design a crypto-friendly alternative protocol called *validity checking* that functionally replaces the normalization of FLTrust. Then, we propose a new two-phase computing paradigm, which effectively tackles the efficiency issue of the directional similarity measurement.

Crypto-friendly protocol for normalization. Intuitively, we can leave the implementation of normalization on the party side in plaintext, since such processing of each local gradient is independent of other gradients. However, under our adversary setting, a key challenge is malicious parties may deviate from the protocol by providing local gradients in the wrong form. Thus, we further

[3] $\text{ReLU}(x)$ is defined as x if $x \geq 0$ and 0 otherwise.

design an efficient *validity checking* protocol to catch malicious parties who deviate from the specification. The main idea is to check whether the squared ℓ_2 norm of each local gradient lies within a certain interval, as follows:

$$flag_i = \mathbf{1}\{|\langle \boldsymbol{g}_i, \boldsymbol{g}_i \rangle - 1| < \epsilon\}, \tag{4}$$

where ϵ is a predefined constant threshold. Currently, we set it empirically based on the gradients obtained before. Moreover, the upper bound of the threshold can be formally analyzed according to mathematical analysis and the Fixed-Point Arithmetic representation. We leave it in the feature work. In particular, $flag_i$ equals 1, if the party correctly normalizes the local gradients; it equals 0, otherwise. Note that we use interval check instead of equality test, since in privacy-preserving FL framework fixed-point encodings are used to represent floating-point gradients at the cost of a small precision loss [36] [34].

New computing paradigm for similarity measurement. We observe that in real-world scenarios most parties (e.g., mobile devices) may only have few computing resources and limited communication bandwidth. However, as a public cloud service provider, SP has advanced computing equipment and extremely high bandwidth. *Our key insight inspired by the above resource asymmetry is that SP can pre-process the heavy cryptographic operations before the parties' local gradients are available, instead of staying idle and waiting for parties to submit local gradients.* To this end, we propose a new computing paradigm for directional similarity measurement, involving two phases, i.e., the *preamble phase* and the *online phase*, which are distinguished according to whether local gradients are available. During the preamble phase, SP performs matrix multiplication preprocessing using the server gradient $\boldsymbol{g}_s$. A detailed procedure is proposed in Section 5. Benefiting from such technique, during the online phase, the cosine similarity measurement can be evaluated securely with negligible computation overhead and zero communication cost. Combining the above improvements, our crypto-friendly FL protocol is shown in Algorithm 1, and it serves as the underlying algorithm for our SecureFL.

Algorithm 1 Crypto-friendly byzantine-robust FL protocol

Input: Each party P_i, $i \in [n]$, with a local dataset $\mathcal{D}_i$; SP with a seed dataset $\mathcal{D}_s$; learning rate η; batch size b; and the number of training iterations $Iter$.
Output: Global model weight $\boldsymbol{\omega}$.

$\boldsymbol{\omega} \leftarrow$ random initialization.
for $iter \in [Iter]$ **do**
  // Training at the party side in parallel
  for $i \in [n]$ **do**
    $\boldsymbol{g}_i = SGD(\boldsymbol{\omega}, \mathcal{D}_i, b)$. *// P_i computes local gradient.*
    Submit $\boldsymbol{g}_i \leftarrow \frac{\boldsymbol{g}_i}{\|\boldsymbol{g}_i\|}$ to SP. *// Local normalization.*
  end for
  // Training at the server side
  $\boldsymbol{g}_s = SGD(\boldsymbol{\omega}, \mathcal{D}_s)$. *// SP computes server gradient.*
  $\boldsymbol{g}_s \leftarrow \frac{\boldsymbol{g}_s}{\|\boldsymbol{g}_s\|}$. *// Normalization.*
  // Robust aggregation
  for $i \in [n]$ **do**
    $flag_i = 1\{|\langle \boldsymbol{g}_i, \boldsymbol{g}_i \rangle - 1| < \epsilon\}$. *// Validity checking.*
    $cos_i = \langle \boldsymbol{g}_i, \boldsymbol{g}_s \rangle$. *// Similarity measure with preprocessing.*
    $TS_i = flag_i \cdot \mathrm{ReLU}(cos_i)$ *// Trust score computation.*
  end for
  $TS = \sum_{i=1}^{n} TS_i$
  $\boldsymbol{g} = \frac{\|\boldsymbol{g}_s\|}{TS} \sum_{i=1}^{n} TS_i \cdot \boldsymbol{g}_i$. *// Weighted aggregation.*
  // Update global model
  Compute $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \boldsymbol{g}$ and broadcast it to all parties.
end for

5 THE SECUREFL FRAMEWORK

In this section, we show the SecureFL framework that adapts the proposed crypto-friendly FL protocol in Algorithm 1 to the privacy-preserving context. At a high level, each party locally trains the local model and normalizes its gradient before sending it to SP. At the same time, SP trains the current global model on the seed dataset to obtain the server gradient, and pre-processes the heavy matrix multiplication. After receiving local gradients from all selected parties, SP and CS engage in 2PC to securely evaluate the robust aggregation protocol utilizing several customized cryptographic protocols. Our complete SecureFL protocol is shown in Table 2.

5.1 The Detailed SecureFL Framework

Before formally demonstrating our framework, we introduce some notations. We denote SP's and CS's shares of the normalized local gradient as $\langle \boldsymbol{g}_i \rangle_0$ and $\langle \boldsymbol{g}_i \rangle_1$, respectively. The servers' rearranged share matrices are denoted as $\langle R \rangle_0 = (\langle \boldsymbol{g}_1 \rangle_0, \langle \boldsymbol{g}_2 \rangle_0, \cdots, \langle \boldsymbol{g}_n \rangle_0)^T$ and $\langle R \rangle_1 = (\langle \boldsymbol{g}_1 \rangle_1, \langle \boldsymbol{g}_2 \rangle_1, \cdots, \langle \boldsymbol{g}_n \rangle_1)^T$, respectively. Note that $R = (\boldsymbol{g}_1, \boldsymbol{g}_2, \cdots, \boldsymbol{g}_n)^T$. Then, we mainly focus on the robust aggregation process. As illustrated in Section 4.2 we divide the implementation of the protocol into the preamble phase and the online phase. In addition, we add an initialization phase to initialize the cryptographic protocols used.

Phase 1: The Initialization Phase

This phase is called once for the entire protocol, where Beaver's multiplication triples and the keypair of PLHE will be generated.

- SP runs $\mathsf{PLHE.KeyGen}(1^k)$ to obtain a public-secret key pair (pk, sk) and sends the public key pk to CS.
- The Beaver's triples generation procedure is called to generate a large number of Beaver's multiplication triples.

Phase 2: The Preamble Phase

This phase can be performed independent of the local gradients, in which the matrix multiplication is pre-processed to facilitate the efficiency of the online phase.

Matrix multiplication preprocessing: As illustrated in Algorithm 1, the center building block of directional similarity measurement is the matrix multiplication between the server gradient vector $\boldsymbol{g}_s$ and the local gradients matrix R. While the Beaver's multiplication procedure can be called to realize such evaluation, the computation and communication overhead is undesirable especially for large-scale inputs. The key idea to tackle this problem is to preprocess costly cryptographic protocols and to speed up robust aggregation operations during the online phase. Specifically, $R\boldsymbol{g}_s = \langle R \rangle_0 \boldsymbol{g}_s + \langle R \rangle_1 \boldsymbol{g}_s$, and $\langle R \rangle_0 \boldsymbol{g}_s$ in the online phase can be computed by SP locally, when $\langle R \rangle_0$ is available. We observe that $\langle R \rangle_1 \boldsymbol{g}_s$ can be computed in the preamble phase, since $\langle R \rangle_1$ can be generated

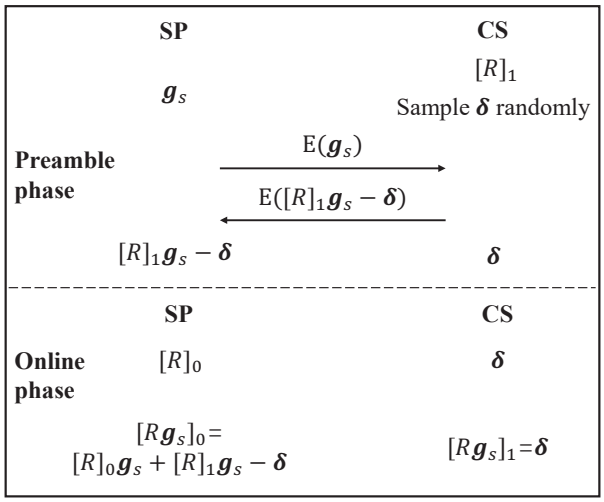


Figure 2: Matrix Multiplication Preprocessing

in advance by CS using PRGs (more details will be given in the online phase) and g_s is known by SP after training the global model. Therefore, similar to [34] [32], we devise a matrix multiplication preprocessing $\mathcal{F}_{\text{matMulPre}}$ by utilizing PLHE, and the protocol is described in Figure 2.

Formally, we have the followings:

- SP sends PLHE.Enc(pk, g_s) to CS. On the input the share matrix $\langle R\rangle_1$, the ciphertext of g_s and a random vector δ, CS computes PLHE.Enc$(pk, \langle R\rangle_1 g_s - \delta)$ using the PLHE.Eval procedure. After that, CS sends this ciphertext to SP.
- SP decrypts the above ciphertexts, to obtain $\langle R\rangle_1 g_s - \delta$. CS holds δ and thus CS and SP hold an additive secret sharing of $\langle R\rangle_1 g_s$. Later, when $\langle R\rangle_0$ becomes available in the online phase, SP computes $\langle R\rangle_0 g_s + \langle R\rangle_1 g_s - \delta$, hence completing the computation of $\langle Rg_s\rangle$.

To enable an efficient implementation of the PLHE.Eval procedure, we take advantage of the matrix-vector multiplication in [27] [34], which effectively employs packed encoding, and SIMD addition and scalar multiplication operations. Note that different from Delphi [34] that operates on a shared vector and a plaintext matrix, our SecureFL executes the operation of multiplying a shared matrix with a plaintext vector, and hence a new protocol is deployed as shown in Figure 2. The security of matrix multiplication preprocessing satisfies the following theorem, and the proof is discussed in Appendix B:

Theorem 1. *The protocol in Figure 2 securely implements matrix multiplication procedure in the semi-honest setting, if packed linearly homomorphic encryption is semantically secure.*

Phase 3: The Online Phase

This phase runs when the parties' local gradients are available. Assuming parties have completed local training and obtained normalized local gradients, the protocol for evaluating the robust aggregation consists of the following steps:

Gradients secret sharing. Each party P_i sends the share of the local gradient $\langle g_i\rangle_0 = g_i - r_i$ to SP, while non-interactively sharing $\langle g_i\rangle_1 = r_i$ with CS by using PRGs, to improve communication efficiency. Specifically, P_i establishes a private seed key k_i^{seed} with CS via the Diffie-Hellman key agreement protocol [15]. After that, CS and P_i jointly sample a same random vector by using PRGs on k_i^{seed}, and hence CS's share of the local gradient $\langle g_i\rangle_1 = r_i$ is generated. Compared with traditional methods in which parties send the shares to both CS and SP, we reduce the bandwidth requirement by half.

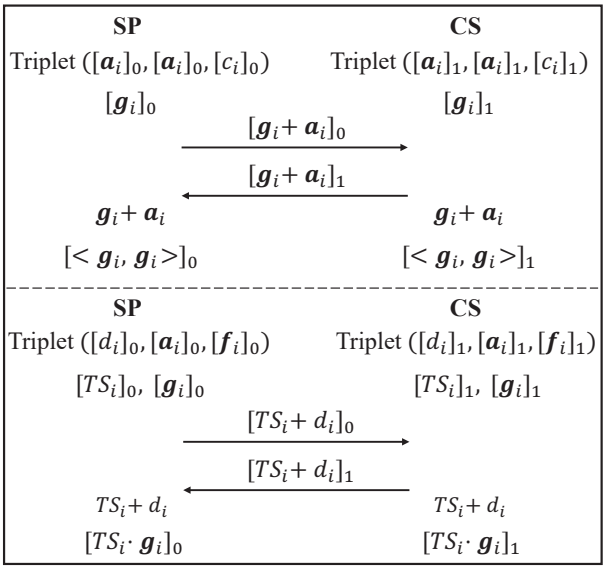


Figure 3: Specialized Beaver Multiplication. The upper part shows the evaluation of the squared ℓ_2 norm in validity check, and the lower part shows the weighted aggregation.

Validity checking. Upon receiving shares of parties' gradients, SP and CS need to check if each party-submitted vector (e.g., size and range) is well formed. As illustrated in Algorithm 1, our devised validity checking mechanism only involves 1 squared ℓ_2 norm, 1 absolute value and 1 compare operations, while the corresponding normalization operation in FLTrust requires 1 squared ℓ_2 norm and d reciprocal square root operations. Note that when being evaluated in existing 2PC libraries [14] [40], the costs of reciprocal square root is much higher than the cost of absolute value and compare. Specifically, we leverage the state-of-the-art algorithm for Millionaires [40] as the building block to construct our compare (DReLU) protocol, as shown in Algorithm 2. Formally, this step is evaluated as follows:

- As shown in the upper part of Figure 3, SP and CS run the Beaver's multiplication procedure[4] to evaluate the squared ℓ_2 norm of g_i. At the end of the procedure, SP holds $\langle \|g_i\|^2\rangle_0$ and CS holds $\langle \|g_i\|^2\rangle_1$. We further observe that the gradient g_i will also be used in weighted aggregation, and hence it suffices to mask it by the same random multiplication triplet (more details will be given later on).

[4]Note that the matrix multiplication preprocessing we designed does not work in such evaluation, since g_i is in the shared form.

- SP and CS run the DReLU procedure in Algorithm 2 to evaluate $flag_i = 1\{|\langle \boldsymbol{g}_i, \boldsymbol{g}_i \rangle - 1| < \epsilon\}$. After that, SP holds $\langle flag_i \rangle_0^B$ and CS holds $\langle flag_i \rangle_1^B$.

Algorithm 2 The protocol of DReLU

Input: SP and CS hold $\langle x \rangle_0$ and $\langle x \rangle_1$, respectively. $\mathcal{F}_{\text{Mill}}$ and $\mathcal{F}_{\text{AND}}$ are adopted from [40] (more details in Appendix C).
Output: SP and CS get $\langle DReLU(x) \rangle_0$ and $\langle DReLU(x) \rangle_1$

1: SP and CS invoke an instance of $\mathcal{F}_{\text{Mill}}$, in which SP's input is $(p - 1 - \langle x \rangle_0)$ and CS's input is $\langle x \rangle_1$. After that SP and CS learn wrap_0 and wrap_1, respectively.
2: SP and CS invoke an instance of $\mathcal{F}_{\text{Mill}}$, in which SP's input is $(p - 1 - \langle x \rangle_0)$ and CS's input is $(p-1)/2 + \langle x \rangle_1$. After that SP and CS learn lwrap_0 and lwrap_1, respectively.
3: SP and CS invoke an instance of $\mathcal{F}_{\text{Mill}}$, in which SP's input is $(p + (p-1)/2 - \langle x \rangle_0)$ and CS's input is $\langle x \rangle_1$. After that SP and CS learn $\langle \text{rwrap} \rangle_0$ and $\langle \text{rwrap} \rangle_1$, respectively.
4: SP and CS invoke an instance of $\mathcal{F}_{\text{AND}}$, in which the inputs of CS and SP are $\langle \text{wrap} \rangle$ and $\langle \text{lwrap} \rangle$, and learn $\langle zl \rangle$.
5: SP and CS invoke an instance of $\mathcal{F}_{\text{AND}}$, in which the inputs of CS and SP are $\langle \text{wrap} \rangle$ and $\langle \text{rwrap} \rangle$, and learn $\langle zr \rangle$.
6: SP and CS output $1 \oplus \langle zl \rangle_0 \oplus \langle zr \rangle_0$ and $\langle zl \rangle_1 \oplus \langle zr \rangle_1$, respectively.

Similar to our *validity checking* setting, Gibbs et al. presented a zero knowledge based input validation protocol (SNIP) [13], where each party needs to generate a SNIP proof which will be used by servers to validate the input. However, the SNIP does not work in our SecureFL, since the proof generation on parties is time-consuming especially with the large dimension of submission, which runs counter to our goal of efficient protocols for parties. Moreover, the result of validation is leaked in their protocol, but rather our SecureFL provides full privacy protection including the result of validity checking.

Cosine similarity computation. Benefiting from the matrix multiplication shares generated in the preamble phase in Figure 2, cosine similarity computations can be non-interactively performed over secret-shared data without invoking heavy cryptographic protocols like OT and PLHE.

- At the beginning of this step, CS holds $\boldsymbol{\delta}$ generated in the preamble phase, and we set $\langle cos_i \rangle_1 = \boldsymbol{\delta}[i], \forall i \in [n]$. Then, SP computes $temp = \langle R \rangle_0 \boldsymbol{g}_s + \langle R \rangle_1 \boldsymbol{g}_s - \boldsymbol{\delta}$, and sets $\langle cos_i \rangle_0 = temp[i], \forall i \in [n]$. As such, the cosine similarity $\langle cos_i \rangle$ between the local gradient $\boldsymbol{g}_i$ and the server gradient $\boldsymbol{g}_s$ is computed, since $cos_i = R\boldsymbol{g}_s[i] = \boldsymbol{g}_i^{\mathrm{T}} \boldsymbol{g}_s$.

Note that an alternative to evaluate this step is to use the Beaver's multiplication technique. However, compared to our preprocessing-based evaluation, for each cosine similarity computation it consumes $(|\boldsymbol{g}_i| + |\boldsymbol{g}_s|)$-bits communication and about 6× more computation overhead during the online phase.

Trust score computation. In this step, the ReLU operation and the boolean-integer multiplication are performed. Specifically, we leverage the DReLU protocol in Algorithm 2 as the building block to construct our ReLU protocol, as shown in Algorithm 3. Besides, we develop an efficient boolean-integer product protocol in Algorithm 4 based on COT techniques, which effectively reduces the communication cost by half.

Algorithm 3 The protocol of ReLU

Input: SP and CS hold $\langle \mathbf{x} \rangle_0$ and $\langle \mathbf{x} \rangle_1$, respectively.
Output: SP and CS get $\langle ReLU(\mathbf{x}) \rangle_0$ and $\langle ReLU(\mathbf{x}) \rangle_1$

1: SP and CS invoke $\mathcal{F}_{\text{DReLU}}$ with input $\langle \mathbf{x} \rangle$ to learn output $\langle \mathbf{y} \rangle^B$.
2: SP and CS invoke $\mathcal{F}_{\text{BmulA}}$ in Algorithm 4 with input $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle^B$ to learn output $\langle \mathbf{z} \rangle$, and sets $\langle ReLU(\mathbf{x}) \rangle = \langle \mathbf{z} \rangle$.

- SP and CS run the ReLU procedure for each $i \in n$, where the inputs are $\langle cos_i \rangle_0$ and $\langle cos_i \rangle_1$, respectively. After that, SP and CS learn $\langle ReLU(cos_i) \rangle_0$ and $\langle ReLU(cos_i) \rangle_1$.
- SP and CS run the boolean-integer multiplication procedure to evaluate $\langle TS_i \rangle = \langle flag_i \rangle^B \cdot \langle ReLU(cos_i) \rangle$. At the end of the procedure, SP holds $\langle TS_i \rangle_0$ and CS holds $\langle TS_i \rangle_1$.

Algorithm 4 Secure Boolean-Integer Multiplication

Input: Additive shares of $a \in \mathbb{Z}_p$ and boolean shares of $b \in \mathbb{Z}_2$
Output: Additive shares of $c = ab \in \mathbb{Z}_p$

1: SP and CS construct correlation functions $f_{cor}^0(x) = x - \langle b \rangle_0^B \langle a \rangle_0 + (1 - \langle b \rangle_0^B) \langle a \rangle_0$ and $f_{cor}^1(x) = x - \langle b \rangle_1^B \langle a \rangle_1 + (1 - \langle b \rangle_1^B) \langle a \rangle_1$, respectively.
2: The parties run $\text{COT}(f_{cor}^0, \langle b \rangle_1^B)$ with SP acting as the sender, and SP obtains x while CS obtains y.
3: The parties run $\text{COT}(f_{cor}^1, \langle b \rangle_0^B)$ with CS acting as the sender, and CS obtains x' while SP obtains y'.
4: SP sets $\langle c \rangle_0 = \langle c \rangle_0^B \langle a \rangle_0 - x + y'$.
5: CS sets $\langle c \rangle_1 = \langle c \rangle_1^B \langle a \rangle_1 - x' + y$.

Weighted aggregation. At the core of this step is a 2PC sub-protocol for computing the weighted aggregation of the normalized local gradients, i.e., evaluating scalar-vector product $TS_i \cdot \boldsymbol{g}_i$. The challenge here is that previous methods evaluating $TS_i \cdot \boldsymbol{g}_i$ require $2ld + 2l$ sent bits, where d is the dimension of each gradient and l is the bit length of each component. To solve such challenge, as shown in Figure 3, we develop a specialized scalar-vector product protocol, inspired by Beaver's triples in the matrix form [36]. The key idea is that the same $\boldsymbol{a}_i$ masking the local gradient $\boldsymbol{g}_i$ in the validity checking can be reused to hide the same $\boldsymbol{g}_i$ in the scalar-vector product evaluation. Its security is guaranteed by the security of the multiplication triples in the matrix form [36]. For completeness, we give a sketch of the security proof in Appendix B. Note that Beaver's triples cannot be used to mask $\boldsymbol{g}_i$'s in different iterations for security. As a result, the bandwidth requirement of our solution is $2l$, an $\frac{2ld+2l}{2l} = d + 1$ improvement. This reduction is nontrivial especially for state-of-the-art neural networks such as ResNets [23], where d is in the order of millions.

- SP and CS run the Beaver's multiplication procedure to evaluate $\langle TS_i \boldsymbol{g}_i \rangle$. At the end of the procedure, SP holds $\langle TS_i \boldsymbol{g}_i \rangle_0$ and CS holds $\langle TS_i \boldsymbol{g}_i \rangle_1$.
- SP and CS compute $\langle \sum_{i \in [n]} TS_i \boldsymbol{g}_i \rangle_0$ and $\langle \sum_{i \in [n]} TS_i \boldsymbol{g}_i \rangle_1$, respectively. After that, CS sends $\langle \sum_{i \in [n]} TS_i \boldsymbol{g}_i \rangle_1$ to SP, which reconstructs $\sum_{i \in [n]} TS_i \boldsymbol{g}_i$.

Remark 2.

(1) *Our SecureFL is robust to parties dropping out.* In many FL systems, participants are mainly composed of resource-constrained

Table 2: The SecureFL Framework

PARAMETERS:

- Number of parties n, dimension of each gradient d. Ideal primitives $\mathcal{F}_{\text{Mult}}$, $\mathcal{F}_{\text{Beaver}}$, $\mathcal{F}_{\text{DReLU}}$, $\mathcal{F}_{\text{ReLU}}$, $\mathcal{F}_{\text{AND}}$ and $\mathcal{F}_{\text{BmulA}}$.

INPUT:

- Each party P_i with local datasets $\mathcal{D}_i$, $i \in [n]$, SP with seed dataset $\mathcal{D}_s$.

PROTOCOL:

I. **Initialization**:
   // *Parties side.*
   a. All parties initialize the architecture F and weights $\boldsymbol{\omega}$ of the global model.
   b. Each party generates a private seed key k_i^{seed} with CS via exchanging Diffie-Hellman public keys and engaging in a key agreement.
   // *Servers side.*
   a. SP runs $\text{PLHE.KeyGen}(1^k) \rightarrow (pk, sk)$ and sends the public key pk to CS.
   b. SP and CS run the Beaver's triples generation protocol $\mathcal{F}_{\text{Beaver}}$ to generate Beaver's multiplication triples.

II. **Training**: Repeat the steps II-IV until the stopping criterion.
   // *Parties side.*
   a. Each party P_i runs $SGD(\boldsymbol{\omega}, \mathcal{D}_i, b) \rightarrow \boldsymbol{g}_i$, and computes the normalized local gradient $\boldsymbol{g}_i \leftarrow \frac{\boldsymbol{g}_i}{\|\boldsymbol{g}_i\|}$.
   b. Each party P_i generates $\boldsymbol{r}_i = \text{PRG}(k_i^{seed})$, sets $\langle \boldsymbol{g}_i \rangle_1 = \boldsymbol{r}_i$ and computes $\langle \boldsymbol{g}_i \rangle_0 = \boldsymbol{g}_i - \boldsymbol{r}_i$.
   // *Servers side.*
   a. SP runs $SGD(\boldsymbol{\omega}, \mathcal{D}_s) \rightarrow \boldsymbol{g}_s$, and computes the normalized server gradient $\boldsymbol{g}_s \leftarrow \frac{\boldsymbol{g}_s}{\|\boldsymbol{g}_s\|}$.
   b. SP computes $E(\boldsymbol{g}_s) = \text{PLHE.Enc}(pk, \boldsymbol{g}_s)$ and sends it to CS.
   c. CS generates $\langle \boldsymbol{g}_i \rangle_1 = \text{PRG}(k_i^{seed})$, $\forall i \in [n]$, and sets $\langle R \rangle_1 = (\langle \boldsymbol{g}_1 \rangle_1, \langle \boldsymbol{g}_2 \rangle_1, \cdots, \langle \boldsymbol{g}_n \rangle_1)^T$.
   d. CS samples a random vector $\boldsymbol{\delta}$, performs $\text{PLHE.Eval}(pk, E(\boldsymbol{g}_s), \langle R \rangle_1) \rightarrow E(pk, \langle R \rangle_1 \boldsymbol{g}_s - \boldsymbol{\delta})$ using the devised matrix multiplication procedure, and sends $E(pk, \langle R \rangle_1 \boldsymbol{g}_s - \boldsymbol{\delta})$ to SP. Besides, CS sets $\langle cos_i \rangle_1 = \boldsymbol{\delta}[i]$, $\forall i \in [n]$.
   e. SP decrypts the above ciphertexts to obtain $\langle R \rangle_1 \boldsymbol{g}_s - \boldsymbol{\delta}$.

III. **Aggregation**:
   // *Parties side.*
   a. Each party P_i shares $\langle \boldsymbol{g}_i \rangle_0 = \boldsymbol{g}_i - \boldsymbol{r}_i$ to SP.
   // *Servers side.*
   a. SP and CS invoke an instance of $\mathcal{F}_{\text{Mult}}$ for each $i \in [n]$, where SP's input is $\langle \boldsymbol{g}_i \rangle_0$ and CS's input is $\langle \boldsymbol{g}_i \rangle_1$. SP and CS learn $\langle \|\boldsymbol{g}_i\|^2 \rangle_0$ and $\langle \|\boldsymbol{g}_i\|^2 \rangle_1$, respectively.
   b. SP and CS invoke two instances of $\mathcal{F}_{\text{DReLU}}$ for each $i \in [n]$, where the inputs are $\langle \|\boldsymbol{g}_i\|^2 \quad \epsilon - 1 \rangle$ and $\langle \epsilon \quad 1 - \|\boldsymbol{g}_i\|^2 \rangle$, respectively. SP and CS learn $\langle flag_{i,0} \rangle^B$ and $\langle flag_{i,1} \rangle^B$, respectively.
   c. SP and CS invoke an instance of $\mathcal{F}_{\text{AND}}$ for each $i \in [n]$, where the inputs are $\langle flag_{i,0} \rangle^B$ and $\langle flag_{i,1} \rangle^B$, respectively. SP and CS learn $\langle flag_i \rangle_0^B$ and $\langle flag_i \rangle_1^B$, respectively.
   d. SP sets $\langle R \rangle_0 = (\langle \boldsymbol{g}_1 \rangle_0, \langle \boldsymbol{g}_2 \rangle_0, \cdots, \langle \boldsymbol{g}_n \rangle_0)^T$, computes $temp = \langle R \rangle_0 \boldsymbol{g} \quad \langle R \rangle_1 \boldsymbol{g} - \boldsymbol{\delta}$, and sets $\langle cos_i \rangle_0 = temp[i]$, $\forall i \in [n]$.
   e. SP and CS invoke an instance of $\mathcal{F}_{\text{ReLU}}$ for each $i \in [n]$, where the inputs are $\langle cos_i \rangle_0$ and $\langle cos_i \rangle_1$, respectively. SP and CS learn $\langle ReLU(cos_i) \rangle_0$ and $\langle ReLU(cos_i) \rangle_1$.
   f. SP and CS invoke an instance of $\mathcal{F}_{\text{BmulA}}$ for each $i \in [n]$, where the inputs are $\langle ReLU(cos_i) \rangle$ and $\langle flag_i \rangle^B$, respectively. SP and CS learn $\langle TS_i \rangle_0$ and $\langle TS_i \rangle_1$.
   g. SP and CS compute $\langle TS \rangle = \sum_{i=1}^n \langle TS_i \rangle$, locally.
   h. SP and CS invoke an instance of $\mathcal{F}_{\text{Mult}}$ for each $i \in [n]$, where the inputs are $\langle \boldsymbol{g}_i \rangle$ and $\langle TS_i \rangle$, respectively. SP and CS learn $\langle TS_i \boldsymbol{g}_i \rangle_0$ and $\langle TS_i \boldsymbol{g}_i \rangle_1$.
   i. SP and CS compute $\langle \boldsymbol{g} \rangle = \sum_{i=1}^n \langle TS_i \boldsymbol{g}_i \rangle$ locally.
   j. CS sends $\langle TS \rangle_1$ and $\langle \boldsymbol{g} \rangle_1$ to SP. After that, SP reconstructs TS and $\boldsymbol{g}$, and computes $\boldsymbol{g}^{global} = \frac{\|\boldsymbol{g}_s\|}{TS} \boldsymbol{g}$.

IV. **Broadcast**:
   // *Servers side.*
   a. SP updates the global weight $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \boldsymbol{g}^{global}$, and broadcasts it to all parties.
   // *Parties side.*
   a. Each party P_i updates its local model by utilizing the global weight $\boldsymbol{\omega}$ received.

mobile devices, so they are likely to drop in each round of aggregation. However, CS non-interactively generates the sharing $\langle R \rangle_1$ of the local gradient matrix by using PRGs, assuming that all parties are online. We resolve this problem by using a simple method, i.e., letting CS remove the corresponding rows in $\langle R \rangle_1$ that are generated for dropped parties, to align with $\langle R \rangle_0$. Note that as the number of dropped parties increases, existing methods such as [10] cause a quadratic communication overhead, but our approach to handling dropped-out parties can actually reduce the overhead of CS and SP (see our experimental results in Tables 4 and 5).

(2) *SecureFL performs fixed-point arithmetic in finite fields.* The implementation of the FL gradient aggregation performs arithmetic operations on floating-point numbers. We work around this by using fixed-point representations of floating-point numbers and embedding them in our finite fields, consistent with existing methods [29] [34]. Furthermore, to prevent values from overflowing due to arithmetic operations, we use the truncation technique from [36]. This method simply truncates the extra LSBs of fixed-point values even when the value is secretly shared, albeit at the cost of a 1-bit error.

(3) *SecureFL can be flexibly extended to support layer-wise robust aggregation.* The discussion so far assumes that the robust aggregation is performed over the entire gradient of each party. However, for modern large-scale neural networks that even contain thousands of layers, we may need to perform the robust aggregation layer-wisely. Concretely, to compute the trust score of each party, we can adaptively assign a weight to each layer (or a combination of several layers). After that, SecureFL is called iteratively for each layer, and the overall trust score is calculated by performing a weighted aggregation of each layer's trust score. This has two advantages: 1) mainly focusing on the more important layers for large-scale models, and 2) preventing overflow when performing the cosine similarity and the squared ℓ_2 norm evaluations.

(4) *The cryptographic recipes of SecureFL may be of general interests.* Given that several recent byzantine-robust FL schemes [9] [16] extensively utilize similarity measurements, our matrix multiplication pre-processing and ReLU-based comparison techniques can be extended to empower the realization of privacy protection. Moreover, the use cases of the validity checking technique go beyond byzantine-robust FL schemes, such as scientific computing [13] and secure querying [12], where servers need to check whether values uploaded by (malicious) users are well-formed in the ciphertext.

5.2 Security analysis

Theorem 2. *The protocol in Table 2 is a cryptographic FL protocol in the honest-but-curious setting, given the ideal primitives of the Beaver's multiplication procedure, ReLU/DReLU and packed linearly homomorphic encryption.*

Proof. We provide a hybrid argument proof in Appendix B that relies on the simulators of the above ideal functionalities.

Theorem 3. *Our SecureFL is byzantine-robust (i.e., can reject incorrectly formatted and poisoned gradients), assuming the soundness of the validity checking construction and the robustness property of our crypto-friendly FL method (directly derived from FLTrust [11]).*

Proof. We provide a brief sketch of the byzantine robustness argument in Appendix B.

6 EVALUATION

We employ two distinct code bases for the implementation of SecureFL. Cryptographic protocols are implemented in C++, relying on the SEAL homomorphic encryption library[5] for PLHE and CrypTFlow library[6] for 2PC protocols. On the other hand, FL experiments are developed in Python and experimental settings mainly follow the previous work [11].

6.1 Experimental Setup

We describe the detailed experimental setup in this section.

Cryptographic setting: To purely measure our SecureFL's performance, we conduct simulations on a Linux machine with an Intel Xeon(R) CPU E5-2620 v4 (2.10 GHz), 16 GB of RAM. Moreover, we compare prior works with our SecureFL in a simulated LAN network.

Datasets and model architectures: Our experiments rely on the HAR, MNIST and CIFAR-10 datasets. A multinomial Logistic Regression (LR) classifier [11] and a shallow convolutional neural network LeNet [30], and a widely used residual neural network ResNet20 [23] are employed as the global model respectively. By default, we set the parties' dataset independent and identically distributed (IID). We also simulate Non-IID dataset using the method of [18] [11]. Details refer to Appendix A.

Evaluated byzantine attacks: We conduct experiments against both data poisoning attacks and local model poisoning attacks. In the former case, we utilize the popular label flipping attack [11], where malicious parties change each sample's label to an arbitrarily wrong label. In the latter case, we use the same attack setting as [18], which is the state-of-the-art local model poisoning attack specific for the Krum aggregation rule.

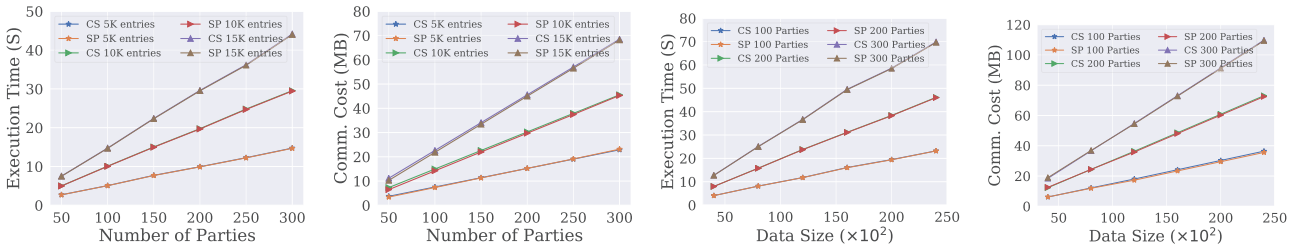
6.2 SecureFL's Cryptographic Protocols

In our evaluations, we omit the cost of the base OTs, and generating the Beaver's multiplicative triples and the keypair of PLHE, as they are a one-time expense during the whole protocol execution.

The performance of SecureFL. Figure 4 reports the execution time and communication cost of SecureFL. From Figures 4(a) and 4(c), we can observe that the execution time of both SP and CS increases linearly with both the number of parties and the number of data entries. The difference between CS and SP is small, which is also reflected in the communication overhead in Figures 4(b) and 4(d). The main reason is that the protocol of our SecureFL is executed between CS and SP and realizes $O(dn)$ overhead, where d denotes the size of gradient and n is the number of parties. This shows SecureFL has excellent scalability despite a large number of parties and data entries involved. Note that we omit communication cost plots for the parties, as they are essentially identical to those for the vanilla FL setting. This is because parties only communicate with SP, and additionally utilize PRGs to non-interactively share secrets (i.e., the sharing of local gradients) with CS. We also report the detailed communication and computational overheads of SecureFL's building blocks in Table 4 and Table 5 of Appendix A.

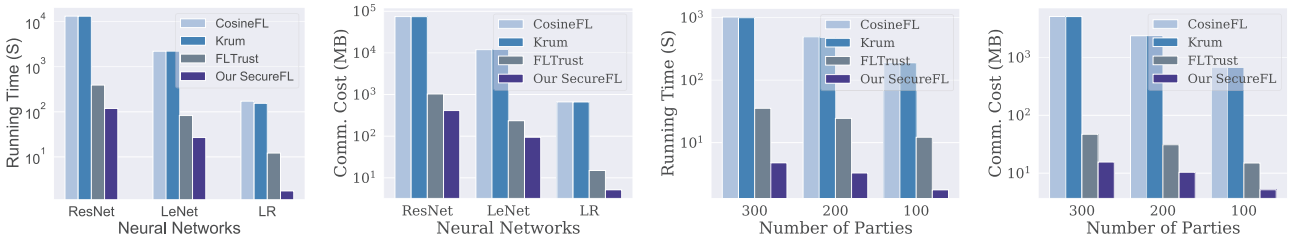
Comparison with prior works. To demonstrate the effectiveness of SecureFL, we implement state-of-the-art private and robust

[5] https://github.com/Microsoft/SEAL
[6] https://github.com/mpc-msri/EzPC/tree/master/SCI



(a) Execution time of CS and SP, as the number of parties increases.
(b) Communicaiton cost of CS and SP, as the number of parties increases.
(c) Execution time of CS and SP, as the size of the data increases.
(d) Communicaiton cost CS and SP, as the size of the data increases.

Figure 4: Execution Time and Communication Cost of SecureFL. In (a)(b), different lines show different data sizes, and in (c)(d), different lines show different numbers of parties. Each point in the figures is the average value over 10 runs.

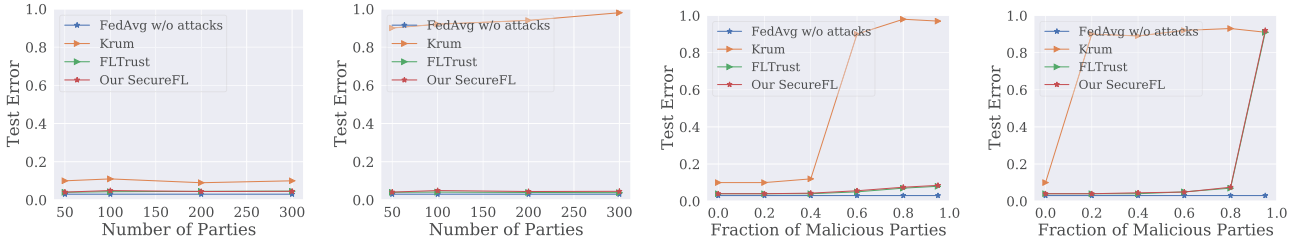


(a) Execution time over the different model architectures.
(b) Communicaiton cost over the different model architectures.
(c) Execution time over the different numbers of parties.
(d) Communicaiton cost over the different numbers of parties.

Figure 5: Execution Time and Communication Cost of SecureFL and Prior Works. In (a)(b), the number of parties is fixed as 100. In (c)(d), LR is used to evaluate. Each point in the figures is the average value over 10 runs and the y-axis is in *log* scale.

Table 3: The Test Error of Different FL Methods under Different Datasets and Various Attacks. The fraction of malicious parties is fixed as 20% and the number of parties is fixed as 100.

	No attack			LabelFlip attack			LocalModel attack		
	HAR	MNIST	CIFAR	HAR	MNIST	CIFAR	HAR	MNIST	CIFAR
FedAvg	0.03	0.04	0.16	0.17	0.06	0.21	0.03	0.10	0.24
Krum	0.12	0.10	0.54	0.10	0.10	0.56	0.22	0.90	0.90
FLTrust	0.04	0.04	0.18	0.04	0.04	0.18	0.04	0.04	0.18
Our SecureFL	**0.03**	**0.04**	**0.19**	**0.03**	**0.04**	**0.19**	**0.04**	**0.04**	**0.19**



(a) Test error under label flipping attack with different numbers of parties.
(b) Test error under local model poisoning attack with different numbers of parties.
(c) Test error under label flipping attack with different fractions of malicious parties.
(d) Test error under local model poisoning attack with different fractions of malicious parties.

Figure 6: Impact of the Number of Parties and the Fraction of Malicious Parties on the Test Error. The MNIST dataset is used to evaluate. In (a)(b), the fraction of malicious parties is fixed as 20% and in (c)(d), the number of parties is fixed as 100.

FL methods including privacy-preserving Krum [24] and FLTrust [11] with generic MPC techniques, as well as secure Cosine distance based FL (called CosineFL) [29]. We compare them with our SecureFL in terms of execution time and communication cost in Figure 5. Note that current implementations of the above FL variants execute top-k[7] under plaintext due to the efficiency issue, but there are serious privacy risks as discussed in Section 2. For making a fair comparison, we implement the top-k protocol (Algorithm 1 in [12]), and utilize it to securely evaluate the above FL variants. In our evaluation, k is fixed as 10%. In addition, we implement the reciprocal squared root protocol in FLTrust by utilizing the CrypTFlow framework [40], which achieves state-of-the-art optimizations of math operators.

Figures 5(a) and 5(b) compare the execution time and communication cost required to securely execute the robust aggregation of prior works and our SecureFL over three datasets and various model architectures. We observe that among the three model architectures with different parameter scales, SecureFL requires 2-97 × less time to evaluate the privately robust aggregation protocol, and 2.5-129 × less communication. This is because matrix multiplication preprocessing and specialized validity checking techniques achieve significant savings in computational overhead and communication cost. Furthermore, Figures 5(c) and 5(d) compare the execution time and communication cost of four FL methods with different numbers of parties over the LR model architecture. In both cases, we observe that as the number of parties increases, the gap between SecureFL and prior methods grows larger. This is because our SecureFL achieves $O(dn)$ computation and communication complexity, while prior works require $O(dn^2)$ complexity. Actually, the privacy-preserving FLTrust also achieves $O(dn)$ complexity, but costly matrix multiplication and reciprocal square root operations reduce its performance. Overall, among the different numbers of parties, SecureFL requires 7-214 × less time to evaluate the privately robust aggregation, and 3-327 × less communication.

6.3 SecureFL's Byzantine-robust Aggregation

In this evaluation, we report the result of SecureFL's robustness, and compare it with existing byzantine-robust FL methods.

Comparison with existing byzantine-robust aggregations. We compare our SecureFL with prior works, including FedAvg [33], Krum [9] and FLTrust [11], which are popular aggregation rules in (robust) FL frameworks. Table 3 shows the test errors of different FL algorithms under three attack settings and three real-world datasets. First, we can observe that SecureFL achieves comparable accuracy to the traditional FedAvg method when there is no attack. Moreover, SecureFL has test errors similar to the state-of-the-art FLTrust with and without byzantine attacks, which indicates that our crypto-friendly variant does *not* sacrifice robustness and inference accuracy. Second, SecureFL is byzantine-robust against both the label flipping attack and the local model poisoning attack. In contrast, existing methods such as FedAvg and Krum are still vulnerable to advanced byzantine attacks. This is because our SecureFL considers both the magnitude and direction of the local gradients to resist existing attacks.

[7] SP selects k local gradients that are more likely submitted by honest parties.

Impact of the number of parties. We report the test errors under different numbers of parties in Figures 6(a) and 6(b). We can observe that as the number of parties increases, SecureFL achieves stable test errors under the label flipping and local model poisoning attacks, which are similar to FedAvg without any attacks. Specifically, the test errors of SecureFL are close to 0.04 under the above attacks. However, under different numbers of parties, the Krum method cannot defend against the label flipping and the local model attacks (giving about 0.1 and 0.9 test errors respectively). Hence, compared with other methods, our SecureFL has excellent byzantine robustness against various attacks.

Impact of the fraction of malicious parties. Figures 6(c) and 6(d) show the test errors of different proportions of malicious parties under the label flipping and local model poisoning attacks. Our SecureFL can tolerate up to 80% of byzantine parties, while realizing the test errors comparable to FedAvg without any attacks. Specifically, when achieving test errors less than 0.1, SecureFL can resist 95% of malicious parties under the label flipping attack and 80% of ones under the local model poisoning attack. However, existing byzantine-robust FL frameworks such as Krum are still vulnerable to attacks despite a small number of malicious parties. For example, under the label flipping attack, the Krum method can only resist 40% of malicious parties, while sacrificing about 5% of test accuracy. It is even worse under the local model poisoning attack, namely that even 10% of malicious parties are still able to completely compromise the training process. Therefore, SecureFL is still byzantine robust against a large number of malicious parties.

7 CONCLUSION

In this paper, we introduce SecureFL, a new federated learning framework that achieves full privacy protection, high scalability and robustness against strong byzantine attacks at the same time. SecureFL employs crypto-friendly FL algorithms and customized cryptographic protocols for enjoying efficiently mathematical operations. Moreover, we evaluate our SecureFL on three real-world datasets and various neural network architectures against two types of latest byzantine attacks, to justify our claim.

The main downside is that SecureFL adapts the scheme of FLTrust to the privacy-preserving context and thus inherits some constraints of the scheme. In particular, it requires a clean seed dataset collected by the service provider. An additional limitation is that our techniques require two non-colluding servers and they do not efficiently scale to the setting of single-server, tolerating malicious clients. Overcoming this limitation would require constructing completely different secure multi-party computation protocols. We leave the improvements for future works.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Shagufta Mehnaz for constructive comments. This work is supported by the National Natural Science Foundation of China under Grants 62020106013, 61972454, 61802051, 61772121, and 61728102, Sichuan Science and Technology Program under Grants 2020JDTD0007 and 2020YFG0298, the Fundamental Research Funds for Chinese Central Universities under Grant ZYGX2020ZB027.

REFERENCES

[1] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. 2019. QUOTIENT: two-party secure neural network training and prediction. In *Proceedings of ACM CCS*. 1231–1247.

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

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of ACM CCS*. 535–548.

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

[5] Moran Baruch, Gilad Baruch, and Yoav Goldberg. 2019. A little is enough: Circumventing defenses for distributed learning. In *Proceedings of NeuIPS*.

[6] Donald Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *Proceedings of CRYPTO*. 420–432.

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

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

[9] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Proceedings of NeurIPS*. 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 *Proceedings of ACM CCS*. 1175–1191.

[11] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. 2021. FLTrust: Byzantine-robust Federated Learning via Trust Bootstrapping. In *Proceedings of NDSS*.

[12] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. 2020. {SANNS}: Scaling up secure approximate k-nearest neighbors search. In *Proceedings of USENIX Security*. 2111–2128.

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

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *Proceedings of NDSS*.

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

[16] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Louis Alexandre Rouault. 2018. The Hidden Vulnerability of Distributed Learning in Byzantium. In *Proceedings of ICML*.

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive* (2012).

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

[19] FeatureCloud. [n.d.]. Transforming health care and medical research with federated learning. https://featurecloud.eu/about/our-vision/.

[20] Wei Gao, Shangwei Guo, Tianwei Zhang, Han Qiu, Yonggang Wen, and Yang Liu. 2021. Privacy-preserving collaborative learning with automatic transformation search. In *Proceedings of CVPR*. 114–123.

[21] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. 2020. Inverting Gradients–How easy is it to break privacy in federated learning?. In *Proceedings of NeurIPS*.

[22] Hanieh Hashemi, Yongqin Wang, Chuan Guo, and Murali Annavaram. 2021. Byzantine-Robust and Privacy-Preserving Framework for FedML. In *ICLR Workshop on Security and Safety in Machine Learning Systems*.

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

[24] Lie He, Sai Praneeth Karimireddy, and Martin Jaggi. 2020. Secure byzantine-robust machine learning. *arXiv:2006.04747* (2020).

[25] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. 2017. Deep models under the GAN: information leakage from collaborative deep learning. In *Proceedings of ACM CCS*. 603–618.

[26] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *Proceedings of CRYPTO*. 145–161.

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of USENIX Security*. 1651–1669.

[28] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2019. Advances and open problems in federated learning. *arXiv:1912.04977* (2019).

[29] Youssef Khazbak, Tianxiang Tan, and Guohong Cao. 2020. MLGuard: Mitigating Poisoning Attacks in Privacy Preserving Distributed Collaborative Learning. In *Proceedings of IEEE ICCCN*. 1–9.

[30] 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* 1, 4 (1989), 541–551.

[31] Beibei Li, Yuhao Wu, Jiarui Song, Rongxing Lu, Tao Li, and Liang Zhao. 2021. DeepFed: Federated Deep Learning for Intrusion Detection in Industrial Cyber-Physical Systems. *IEEE Transactions on Industrial Informatics* 17, 8 (2021), 5615–5624.

[32] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of ACM CCS*. 619–631.

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

[34] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *Proceedings of USENIX Security*. 2505–2522.

[35] Payman Mohassel, Mike Rosulek, and Ni Trieu. 2020. Practical privacy-preserving k-means clustering. *Proceedings on Privacy Enhancing Technologies* 2020, 4 (2020), 414–433.

[36] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *Proceedings of IEEE S&P*. 19–38.

[37] 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 *Proceedings of S&P*. 739–753.

[38] Thien Duc Nguyen, Phillip Rieger, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Ahmad-Reza Sadeghi, Thomas Schneider, et al. 2021. FLGUARD: Secure and Private Federated Learning. *arXiv preprint arXiv:2101.02281* (2021).

[39] Sundar Pichai. 2019. Privacy should not be a luxury good. *The New York Times* (2019).

[40] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-party secure inference. In *Proceedings of ACM CCS*. 325–342.

[41] Virat Shejwalkar and Amir Houmansadr. 2021. Manipulating the Byzantine: Optimizing Model Poisoning Attacks and Defenses for Federated Learning. In *Proceedings of NDSS*.

[42] Nigel P Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, codes and cryptography* 71, 1 (2014), 57–81.

[43] Jinhyun So, Başak Güler, and A Salman Avestimehr. 2020. Byzantine-resilient secure federated learning. *IEEE Journal on Selected Areas in Communications* (2020).

[44] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *Proceedings of USENIX Security*. 991–1008.

[45] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. 2020. Fall of empires: Breaking Byzantine-tolerant SGD by inner product manipulation. In *Uncertainty in Artificial Intelligence*. 261–270.

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

[47] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. 2018. Applied federated learning: Improving google keyboard query suggestions. *arXiv:1812.02903* (2018).

[48] Andrew C Yao. 1982. Theory and application of trapdoor functions. In *Proceedings of FOCS*. 80–91.

[49] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. 2018. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of ICML*. 5650–5659.

[50] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. 2020. Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning. In *Proceedings of USENIX ATC*. 493–506.

[51] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy ComputAtion for Linear Algebra in Privacy-Preserved Neural Networks. In *Proceedings of NDSS*.

[52] Ligeng Zhu, Zhijian Liu, and Song Han. 2019. Deep Leakage from Gradients. In *Proceedings of NeurIPS*.

A ADDITIONAL EXPERIMENTAL RESULTS

A.1 Performance of SecureFL's building blocks

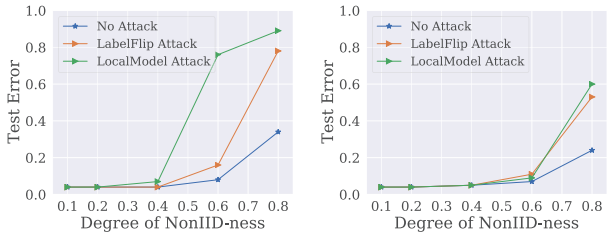
In Tables 4 and 5, we plot in detail the communication and computational overhead of each step of SecureFL.

As shown in Table 4, the validity checking and cosine computation steps occupy most of the computation overhead. With devised matrix multiplication preprocessing in the preamble phase, the online overhead of cosine similarity computations only requires less than 13 ms. For the weighted aggregation step, it only takes less than 230 ms, since the same multiplication triples generated in the validity checking phase will be reused in this step. Beneficially, compared with traditional methods that generate fresh masks for each calculation, the cost of the validity checking and weighted aggregation steps is reduced by about half. Besides, our SecureFL is fully robust to party dropping, where the computational overhead decreases proportionally as the number of dropped parties increases. Below we focus on the communication performance of SecureFL. In Table 5, the validity checking step produces relatively high communications compared to other operations. Nevertheless, similar to the execution time we discussed above, it significantly saves the cost of the weighted aggregation step by reusing the same masks to local gradients, i.e., less than 1 KB under 100 parties setting. More importantly, in the cosine similarity evaluation, the communication overhead is low and remains constant as the number of parties increases. Besides, as the number of dropped parties increases, the communication overhead decreases proportionally.

A.2 Additional Robustness Evaluations

We simulated Non-IID dataset using the method of [18] [11]. Specifically, assuming M classes in the classification task, we evenly split the parties into M groups. We assign a sample with label m to the m-th group with probability β, and to any other group with probability $\frac{1-\beta}{M-1}$. Informally, the value of β controls the degree of Non-IID-ness. $\beta = \frac{1}{M}$ represents IID data distributions, and the higher β is, the more likely the parties hold samples from only one class.

We evaluate the impact of the degree of Non-IID-ness on the test error on MNIST dataset. The fraction of malicious parties is fixed as 20% and the number of parties is fixed as 100. Figure 7(a) shows, when the degree of Non-IID-ness of seed dataset varies, the test error of our method under different attacks, i.e., no attack, label flipping attack and local model poisoning attack. We observe that SecureFL is accurate and robust when the degree of Non-IID-ness is not significant. In particular, when $\beta \leq 0.4$ for seed dataset, SecureFL realizes the test errors comparable to FedAvg without any attacks. Besides, we also study the impact of the Non-IID parties' local datasets in Figure 7(b). Our results show that when $\beta \leq 0.6$ for parties' datasets, SecureFL achieves excellent performance, i.e., the test error is less than 0.1. Therefore, SecureFL works well when the parties' data distribution does not differ too much.



(a) Test error under Non-IID root dataset. (b) Test error under Non-IID parties' local dataset.

Figure 7: Impact of the Degree of Non-IID-ness on the Test Error.

B SECURITY PROOFS

B.1 Proof of Theorem 1

Proof. Our security proof follows the ideal-world/real-world paradigm: in real-world, SP and CS interact according to the protocol specification, whereas in ideal-world they have access to a ideal functionality $\mathcal{F}_{\text{matMulPre}}$. The executions in both worlds are coordinated by the environment Env, who chooses the inputs to CS and SP and plays the role of a distinguisher between the real and ideal executions. We will show that the real-world distribution is computationally indistinguishable to the ideal-world distribution.

Proof of indistinguishability with corrupted SP. Below, we first construct an ideal-world simulator Sim that performs as follows:

- Sim receives $\boldsymbol{g}_s$ from the environment Env, and sends it to $\mathcal{F}_{\text{matMulPre}}$ and gets the result $\boldsymbol{u}'$.
- Sim encrypts $\boldsymbol{u}'$ using SP's public key and returns $\tilde{\boldsymbol{u}}' = \text{PLHE.Enc}(pk, \boldsymbol{u}')$ to SP.
- Sim outputs whatever SP outputs.

Then, we show that the view Sim simulates for SP is indistinguishable from the view of SP interacting in the real execution. The message $\text{PLHE.Dec}(\tilde{\boldsymbol{u}}')$ is same as $\boldsymbol{u}$ in real execution, where $\boldsymbol{u} = \langle R \rangle_1 \boldsymbol{g}_s - \boldsymbol{\delta}$. Thus, they are indistinguishable even if the private key sk is observed. In addition, it does not reveal any information about $\langle R \rangle_1$ from $\boldsymbol{u}$, since $\boldsymbol{\delta}$ is randomly chosen. Therefore, we claim that the output distribution of Env in real-world is computationally indistinguishable from that in ideal-world.

Proof of indistinguishability with corrupted CS. Below, we first construct an ideal-world simulator Sim that performs as follows:

- Sim receives $\langle R \rangle_1$ and $\boldsymbol{\delta}$ from the environment Env, and sends it to $\mathcal{F}_{\text{matMulPre}}$.
- Sim constructs $\tilde{\boldsymbol{g}_s}' \leftarrow \text{PLHE.Enc}(pk, \boldsymbol{0})$, and gives it to CS.
- Sim outputs whatever CS outputs.

Then, we show that the view $\tilde{\boldsymbol{g}_s}'$ *Sim* simulates for CS is indistinguishable from the view $\text{PLHE.Enc}(pk, \boldsymbol{g}_s)$ of CS interacting in the real execution, because of the semantic security of PLHE. Thus, the output distribution of Env in real-world is computationally indistinguishable from that in ideal-world.

□

B.2 Proof of Theorem 2

Proof. We construct a simulator Sim simulates the view of corrupted SP, which consists of her input/output and received messages. The simulator for CS should be the same. *Sim* proceeds as follows:

- In the validity checking, Sim calls simulators $\text{Sim}_{\mathcal{F}_{\text{Mult}}}(\langle \boldsymbol{g}_i \rangle)$, $\text{Sim}_{\mathcal{F}_{\text{DReLU}}}(\langle \|\boldsymbol{g}_i\|^2 \rangle)$ and $\text{Sim}_{\mathcal{F}_{\text{AND}}}(\langle flag_{i,0} \rangle^B, \langle flag_{i,1} \rangle^B)$ for each $i \in [n]$, and appends their output to the general view.
- In the cosine similarity computation, Sim calls $\mathcal{F}_{\text{matMulPre}}$ simulator $\text{Sim}_{\mathcal{F}_{\text{matMulPre}}}(\langle R \rangle_1, \boldsymbol{g}_s)$, and computes the matrix

Table 4: Execution Time of Each Step of SecureFL. The data size is fixed as 10K entries. Different rows show different numbers of users/dropped users. The values in "CosineComp" column represent preamble/online costs.

UsersNum	Dropout	SharingGen	ValidityCheck	CosineComp	TrustScore	WeightedAgg	TotalTime
100	0%	20 ms	9730 ms	6804 ms / 10 ms	92 ms	67 ms	16723 ms
100	10%	20 ms	8739 ms	6244 ms / 10 ms	88 ms	62 ms	15163 ms
100	20%	20 ms	7725 ms	5637 ms / 9 ms	83 ms	54 ms	13528 ms
300	0%	61 ms	31530 ms	16219 ms / 12 ms	98 ms	230 ms	48150 ms
300	10%	60 ms	26213 ms	14735 ms / 13 ms	95 ms	213 ms	41329 ms
300	20%	61 ms	24033 ms	12929 ms / 11 ms	93 ms	190 ms	37317 ms

Table 5: Communication Cost of Each Step of SecureFL. The data size is fixed as 10K entries. Different rows show different numbers of users/dropped users. The values in "CosineComp" column represent preamble/online costs.

UsersNum	Dropout	SharingGen	ValidityCheck	CosineComp	TrustScore	WeightedAgg	TotalComm
100	0%	0 KB	15689.2 KB	512.1 KB / 0 KB	66.4 KB	0.8 KB	16268.5 KB
100	10%	0 KB	14124.7 KB	512.1 KB / 0 KB	65.2 KB	0.7 KB	14702.7 KB
100	20%	0 KB	12558.3 KB	512.1 KB / 0 KB	62.8 KB	0.6 KB	13133.8 KB
300	0%	0 KB	46988.7 KB	512.1 KB / 0 KB	109.1 KB	2.3 KB	47612.2 KB
300	10%	0 KB	42293.2 KB	512.1 KB / 0 KB	104.2 KB	2.1 KB	42911.6 KB
300	20%	0 KB	37597.9 KB	512.1 KB / 0 KB	87.4 KB	1.9 KB	38199.3 KB

multiplication in the online phase, followed by appending its output to the view.
- In the trust score computation, Sim first calls $\mathcal{F}_{\text{ReLU}}$ simulator $\text{Sim}_{\mathcal{F}_{\text{ReLU}}}(\langle cos_i \rangle)$ for each $i \in [n]$, and then simulates TS_i computation by calling $\text{Sim}_{\mathcal{F}_{\text{BmulA}}}(\langle ReLU(cos_i) \rangle, \langle flag_i \rangle^B)$ for each $i \in [n]$.
- In the weighted aggregation, Sim calls $\mathcal{F}_{\text{Mult}}$ simulator $\text{Sim}_{\mathcal{F}_{\text{Mult}}}(\langle ReLU(cos_i) \rangle, \langle flag_i \rangle^B)$ for each $i \in [n]$, and sums parties' trust scores.

We show that the real world distribution is computationally indistinguishable to the simulated distribution via a hybrid argument. For this, we formally show the simulation by proceeding the sequence of hybrid arguments, $H_0, \cdots, H_4$, where H_0 is the real view and H_4 is the simulated view generated by Sim.

- Hybrid_1: Let H_1 be the same as H_0, except the followings. First, for the squared ℓ_2 norm evaluation in the validity checking phase, we use the simulator $\text{Sim}_{\mathcal{F}_{\text{Mult}}}(\langle \boldsymbol{g}_i \rangle)$. Then the $\mathcal{F}_{\text{DReLU}}$ and $\mathcal{F}_{\text{AND}}$ evaluations are replaced with the simulators $\text{Sim}_{\mathcal{F}_{\text{DReLU}}}(\langle \|\boldsymbol{g}_i\|^2 \rangle)$ and $\text{Sim}_{\mathcal{F}_{\text{AND}}}(\langle flag_{i,0} \rangle^B, \langle flag_{i,1} \rangle^B)$, respectively. Given the simulation security of these protocols, H_1 is indistinguishable from H_0.
- Hybrid_2: Let H_2 be the same as H_1, except the $\mathcal{F}_{\text{matMulPre}}$ execution is replaced with execution the $\text{Sim}_{\mathcal{F}_{\text{matMulPre}}}(\langle R \rangle_1, \boldsymbol{g}_s)$. Because $\mathcal{F}_{\text{matMulPre}}$ is guaranteed to produce output indistinguishable from that of real world, H_2 and H_1 are indistinguishable. Note that the matrix multiplication in the online phase is evaluated non-interactively, thus the view is not changed.
- Hybrid_3: In this hybrid, for the boolean-integer multiplication evaluation and ReLU evaluation, we use the simulators $\text{Sim}_{\mathcal{F}_{\text{ReLU}}}(\langle cos_i \rangle)$ and $\text{Sim}_{\mathcal{F}_{\text{BmulA}}}(\langle ReLU(cos_i) \rangle, \langle flag_i \rangle^B)$. It follows from simulation security that H_3 is indistinguishable from H_2.
- Hybrid_4: Let H_4 be the same as H_3, except the weighted aggregation evaluation is replaced with execution the simulator $\text{Sim}_{\mathcal{F}_{\text{Mult}}}(\langle ReLU(cos_i) \rangle, \langle flag_i \rangle^B)$. Because $\text{Sim}_{\mathcal{F}_{\text{Mult}}}$ is guaranteed to produce output indistinguishable from real, H_4 and H_3 are indistinguishable.

In summary, H_4 is identically distributed to the simulator's output, completing the proof.

□

B.3 Proof of Theorem 3

Proof. The robustness of SecureFL is guaranteed by the soundness of the validity checking and the robustness of FLTrust. Specifically, in the validity checking, CS and SP check each party's input $\boldsymbol{g}_i$, and assign the result to $flag_i$. $flag_i$ concludes that either the servers accept $\boldsymbol{g}_i$ ($flag_i = 1$) or the servers reject $\boldsymbol{g}_i$ ($flag_i = 0$). Thus, any misbehaving party must either submit a well-formed submission or be treated as zero input. On the other hand, to further exclude wrong inputs that are actually in the required domain, our SecureFL employs the byzantine-robust aggregation rule, whose robustness guarantee directly relies on the FLTrust's protocol (the proof is in Section IV-D of [11]). It preserves the accuracy of the global model, even when a part of malicious clients perform strong byzantine attacks. In summary, our SecureFL is robust.

□

B.4 Proof of Specialized Beaver's Triplets

Proof. We give a sketch of the security proof here. Assume two servers aim to compute $\langle x \rangle \cdot \langle y \rangle$ and $\langle x \rangle \cdot \langle z \rangle$. The Beaver's triplets are given, i.e., $\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle b' \rangle, \langle c' \rangle$. Without loss of generality, we

Table 6: The ideal functionality $\mathcal{F}_{\text{BmulA}}$

Input:
- SP: $\langle x\rangle_0^B \in \mathbb{Z}_2, \langle y\rangle_0 \in \mathbb{Z}_p$.
- CS: $\langle x\rangle_1^B \in \mathbb{Z}_2$, and $\langle y\rangle_1, r \in \mathbb{Z}_p$.

Output:
- SP: $\langle z\rangle_0 = x \cdot y - r \mod p$ where $x = \langle x\rangle_0^B \oplus \langle x\rangle_1^B$ and $y = \langle y\rangle_0 \quad \langle y\rangle_1 \mod p$.
- CS: $\langle z\rangle_1 = r$.

assume SP is corrupted by an adversary, since the protocol is symmetric with respect to the two servers. The message of SP received from CS is $\langle x\rangle_1 + \langle a\rangle_1$, $\langle y\rangle_1 + \langle b\rangle_1$, and $\langle z\rangle_1 + \langle b'\rangle_1$. Because the triplets are randomly sample from the field, the above messages are indistinguishable from random values. Therefore, the specialized Beaver's multiplication is secure for any adversary.

Algorithm 5 Beaver's Triples Generation

Input: Additive shares of random vectors $\langle \mathbf{a}\rangle \in \mathbb{Z}_p^n$ and $\langle \mathbf{b}\rangle \in \mathbb{Z}_p^n$
Output: Additive shares of vector $\langle \mathbf{c}\rangle \in \mathbb{Z}_p^n$, where $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$

1: SP computes $ct_a \leftarrow \mathsf{Enc}(pk, \langle \mathbf{a}\rangle_0)$, $ct_b \leftarrow \mathsf{Enc}(pk, \langle \mathbf{b}\rangle_0)$, and sends ct_a, ct_b to CS.
2: CS samples a random vector $\mathbf{r} \in \mathbb{Z}_p^n$ and computes $ct_r \leftarrow \mathsf{Enc}(pk, \mathbf{a})$. Then, she homomorphically evaluates $ct'_a \leftarrow \mathsf{Enc}(pk, \langle \mathbf{a}\rangle_0 \cdot \langle \mathbf{b}\rangle_1)$ and $ct'_b \leftarrow \mathsf{Enc}(pk, \langle \mathbf{b}\rangle_0 \cdot \langle \mathbf{a}\rangle_1)$, along with $ct_d \leftarrow \mathsf{Enc}(pk, \langle \mathbf{b}\rangle_0 \cdot \langle \mathbf{a}\rangle_1 + \langle \mathbf{a}\rangle_0 \cdot \langle \mathbf{b}\rangle_1 + r)$. After that, CS sets $\langle \mathbf{c}\rangle_1 = \langle \mathbf{a}\rangle_1 \cdot \langle \mathbf{b}\rangle_1 - \mathbf{r}$, and sends ct_d to SP.
3: SP decrypts the above ciphertexts $\mathbf{d} \leftarrow \mathsf{Dec}(sk, ct_d)$, and obtains $\langle \mathbf{c}\rangle_0 = \langle \mathbf{a}\rangle_0 \cdot \langle \mathbf{b}\rangle_0 + \mathbf{d}$.

Algorithm 6 Beaver's Multiplication Procedure

Input: Additive shares of inputs $\langle \mathbf{x}\rangle \in \mathbb{Z}_p^n$ and $\langle \mathbf{y}\rangle \in \mathbb{Z}_p^n$, along with Beaver's triples $\langle \mathbf{a}\rangle \in \mathbb{Z}_p^n$, $\langle \mathbf{b}\rangle \in \mathbb{Z}_p^n$ and $\langle \mathbf{c}\rangle \in \mathbb{Z}_p^n$
Output: Additive shares of vector $\langle \mathbf{z}\rangle \in \mathbb{Z}_p^n$, where $\mathbf{z} = \mathbf{x} \cdot \mathbf{y}$

1: SP and CS first set $\langle \mathbf{e}\rangle = \langle \mathbf{x}\rangle - \langle \mathbf{a}\rangle \mod p$ and $\langle \mathbf{f}\rangle = \langle \mathbf{y}\rangle - \langle \mathbf{b}\rangle \mod p$
2: SP and CS reconstruct $\mathbf{e}$ and $\mathbf{f}$ interactively.
3: SP sets $\langle \mathbf{z}\rangle_0 = \mathbf{f} \cdot \langle \mathbf{a}\rangle_0 + \mathbf{e} \cdot \langle \mathbf{b}\rangle_0 + \langle \mathbf{c}\rangle_0$ and CS sets $\langle \mathbf{z}\rangle_1 = \mathbf{e} \cdot \mathbf{f} + \mathbf{f} \cdot \langle \mathbf{a}\rangle_1 + \mathbf{e} \cdot \langle \mathbf{b}\rangle_1 + \langle \mathbf{c}\rangle_1$.

C FUNCTIONALITIES AND PROTOCOLS

We introduce the ideal primitives and protocols used by SecureFL.

Boolean-integer multiplication. The functionality $\mathcal{F}_{\text{BmulA}}$ of boolean-integer multiplication (Algorithm 4 shows the protocol) is described in Table 6. Security trivially follows 1-out-of-2 OTs.

Beaver's multiplication procedure. Given two shared values $\langle x\rangle$, $\langle y\rangle$ and constant c, linear operations such as $\langle cx + y\rangle$ can be directly computed by parties via locally adding the shares. In addition, multiplication operations on shared values can be implemented by

Table 7: The ideal functionality $\mathcal{F}_{\text{DReLU}}$

Input:
- SP: $\langle x\rangle_0 \in \mathbb{Z}_p$.
- CS: $\langle x\rangle_1 \in \mathbb{Z}_p, r \in \mathbb{Z}_2$.

Output:
- SP: $\langle z\rangle_0^B = compare(x, 0) \oplus r$ where $x = \langle x\rangle_0 \quad \langle x\rangle_1 \mod p$.
- CS: $\langle z\rangle_1^B = r$.

Table 8: The ideal functionality $\mathcal{F}_{\text{ReLU}}$

Input:
- SP: $\langle x\rangle_0 \in \mathbb{Z}_p$.
- CS: $\langle x\rangle_1, r \in \mathbb{Z}_p$.

Output:
- SP: $\langle z\rangle_0 = \text{DReLU}(x) \cdot x - r \mod p$ where $x = \langle x\rangle_0 \quad \langle x\rangle_1 \mod p$.
- CS: $\langle z\rangle_1 = r$.

calling Beaver's multiplicative triples [6]. The ideal functionality $\mathcal{F}_{\text{Beaver}}$ is: sampling a, b from $\mathbb{Z}_p$ uniformly at random, and returning $\langle a\rangle_0, \langle b\rangle_0, \langle ab\rangle_0$ to the first party and $\langle a\rangle_1, \langle b\rangle_1, \langle ab\rangle_1$ to the second party. After generating the Beaver's multiplicative triples, two parties holding $\langle x\rangle$, $\langle y\rangle$ can implement multiplication operations $\mathcal{F}_{\text{Mult}}$ and receive as the output the shares of $\langle xy\rangle$ at the end of the protocol. The protocols of $\mathcal{F}_{\text{Mult}}$ and $\mathcal{F}_{\text{Beaver}}$ are shown in Algorithm 5 and Algorithm 6. In addition, the functionality $\mathcal{F}_{\text{AND}}$ implements that on input $\langle x\rangle^B, \langle y\rangle^B$, SP and CS learn $\langle z\rangle_0^B$ and $\langle z\rangle_1^B$, where $z = x \oplus y$. The protocol of $\mathcal{F}_{\text{AND}}$ can be realized using bit-triples (a boolean variant of the Beaver's triples) via an instance of 1-out-of-16 OTs. We omit a detailed description of the protocol due to space limits.

DReLU and ReLU. The functionalities $\mathcal{F}_{\text{DReLU}}$ and $\mathcal{F}_{\text{ReLU}}$ of DReLU/ReLU operations are discussed in Tables 7 and 8. For a finite field $\mathbb{Z}_p$, $DReLU(x) = 1$ if $x < \lceil p/2 \rceil$ and 0 otherwise. Let arithmetic shares of x be $\langle x\rangle_0$ and $\langle x\rangle_1$. We define wrap $= 1\{\langle x\rangle_0 + \langle x\rangle_1 > p - 1\}$, lwrap $= 1\{\langle x\rangle_0 + \langle x\rangle_1 > (p-1)/2\}$ and rwrap $= 1\{\langle x\rangle_0 + \langle x\rangle_1 > p + (p-1)/2\}$. Then, $DReLU(x)$ is $(1 \oplus \text{lwrap})$ if wrap $= 0$, else it is $(1 \oplus \text{rwrap})$ if wrap $= 1$. The detailed protocols for realizing $\mathcal{F}_{\text{DReLU}}$ and $\mathcal{F}_{\text{ReLU}}$ are discussed in Algorithms 2 and 3, where we utilize the millionaire functionality $\mathcal{F}_{\text{Mill}}$ of [40]. We omit a detailed description of the protocol for realizing $\mathcal{F}_{\text{Mill}}$ due to space limits.