EIFFeL: Ensuring Integrity For Federated Learning

Amrita Roy Chowdhury[*]
University of Wisconsin-Madison

Chuan Guo
Meta AI

Somesh Jha[†]
University of Wisconsin-Madison

Laurens van der Maaten
Meta AI

Abstract

Federated learning (FL) enables clients to collaborate with a server to train a machine learning model. To ensure privacy, the server performs secure aggregation of updates from the clients. Unfortunately, this prevents verification of the well-formedness (integrity) of the updates as the updates are masked. Consequently, malformed updates designed to poison the model can be injected without detection. In this paper, we formalize the problem of ensuring *both* update privacy and integrity in FL and present a new system, EIFFeL, that enables secure aggregation of *verified* updates. EIFFeL is a general framework that can enforce *arbitrary* integrity checks and remove malformed updates from the aggregate, without violating privacy. Our empirical evaluation demonstrates the practicality of EIFFeL. For instance, with 100 clients and 10% poisoning, EIFFeL can train an MNIST classification model to the same accuracy as that of a non-poisoned federated learner in just 2.4s per iteration.

CCS Concepts

• **Security and privacy** → **Cryptography**; **Privacy-preserving protocols**.

Keywords

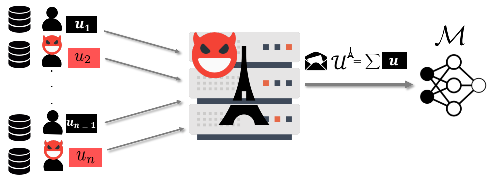
Poisoning Attacks, Input Integrity, Secure Aggregation

ACM Reference Format:

Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. 2022. EIFFeL: Ensuring Integrity For Federated Learning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560611

1 Introduction

Federated learning (FL; [50]) is a learning paradigm for decentralized data in which multiple clients collaborate with a server to train a machine-learning (ML) model. Each client computes an update on its *local* training data and shares it with the server; the server aggregates the local updates into a *global* model update. This allows the clients to contribute to model training without sharing their private data. However, the local updates can still reveal information about a client's private data [11, 52, 53, 76, 78]. FL addresses this by using *secure aggregation*: clients mask the updates they share, and the server can recover *only* the final aggregate in the clear.



Security Goal	Cryptographic Primitive
Input Privacy	Shamir's Threshold Secret Sharing Scheme [60]
Input Integrity	Secret-Shared Non-Interactive Proof [24] Verifiable Secret Shares [30]

Figure 1: EIFFeL performs secure aggregation of *verified* inputs in FL. The table lists its security goals and the cryptographic primitives we adopt to achieve them.

A major challenge in FL is that it is vulnerable to Byzantine attacks. In particular, malicious clients can inject poisoned updates into the learner with the goal of reducing the global model accuracy [10, 12, 29, 36, 51] or implanting backdoors in the model that can be exploited later [5, 21, 71]. Even a single malformed update can significantly alter the trained model [14]. Thus, ensuring the well-formedness of the updates, *i.e.*, upholding their *integrity*, is essential for ensuring robustness in FL. This problem is especially challenging in the context of secure aggregation as the individual updates are masked from the server, which prevents audits on them.

These challenges in FL lead to the research question: *How can a federated learner efficiently verify the integrity of clients' updates without violating their privacy?*

We formalize this problem by proposing *secure aggregation of verified inputs* (SAVI) protocols that: (1) securely verify the integrity of each local update, (2) aggregate *only* well-formed updates, and (3) release only the final aggregate in the clear. A SAVI protocol allows for checking the well-formedness of updates *without observing them*, thereby ensuring *both* the privacy and integrity of updates.

We demonstrate the feasibility of SAVI by proposing EIFFeL: a system that instantiates a SAVI protocol that can perform *any integrity check that can be expressed as an arithmetic circuit with public parameters.* This provides EIFFeL the flexibility to implement a plethora of modern ML approaches that ensure robustness to Byzantine attacks

[*] Work done during internship at Meta AI
[†] Employed part-time at Meta during this work

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560611

by checking the integrity of per-client updates before aggregating them [5, 27, 44, 61, 67, 68, 73, 74]. EIFFeL is a general framework that empowers a federated learner to deploy (multiple) *arbitrary* integrity checks of their choosing on the "masked" updates.

EIFFeL uses secret-shared non-interactive proofs (SNIP; [24]) which are a type of zero-knowledge proofs that are optimized for the client-server setting. SNIP, however, requires multiple honest verifiers to check the proof. EIFFeL extends SNIP to a *malicious* threat model by carefully *co-designing its architectural and cryptographic components*. Moreover, we develop a suite of optimizations that improve EIFFeL's performance by at least 2.3×. Our empirical evaluation of EIFFeL demonstrates its practicality for real-world usage. For instance, with 100 clients and a poisoning rate of 10%, EIFFeL can train an MNIST classification model to the same accuracy as that of a non-poisoned federated learner in just 2.4*s* per iteration.

2 Problem Overview

In this section, we introduce the problem setting, followed by its threat analysis and an overview of our solution.

2.1 Problem Setting

In FL, multiple parties with distributed data jointly train a *global model*, $\mathcal{M}$, without explicitly disclosing their data to each other. FL has two types of actors:

- **Clients.** There are n clients where each client, $C_i, i \in [n]$, owns a private dataset, D_i. The raw data is never shared, instead, every client computes a local update for $\mathcal{M}$, such as the average gradient, over the private dataset D_i.
- **Server.** There is a single *untrusted* server, $\mathcal{S}$, who coordinates the updates from different clients to train $\mathcal{M}$.

A single training iteration in FL consists of the following steps:

- **Broadcast.** The server broadcasts the current parameters of the model $\mathcal{M}$ to all the clients.
- **Local computation.** Each client C_i locally computes an update, u_i, on its dataset D_i.
- **Aggregation.** The server $\mathcal{S}$ collects the client updates and aggregates them, $\mathcal{U} = \sum_{i \in [n]} u_i$.
- **Global model update.** The server $\mathcal{S}$ updates the global model $\mathcal{M}$ based on the aggregated update $\mathcal{U}$.

In settings where there is a large number of clients, it is typical to subsample a small subset of clients to participate in a given iteration. We assume n to denote the number of clients that participate in each iteration and C denotes the subset of these n clients, which the server announces at the beginning of the iteration.

2.2 Security Goals

- **Input Privacy (Client's Goal).** The first goal is to ensure privacy for all *honest* clients. That is, no party should be able learn anything about the raw input (update) u_i of an honest client C_i, other than what can be learned from the final aggregate $\mathcal{U}$.
- **Input Integrity (Server's Goal).** The server $\mathcal{S}$ is motivated to ensure that the individual updates from each client are well-formed. Specifically, the server has a *public* validation predicate, $\mathsf{Valid}(\cdot)$, that defines a syntax for the inputs (updates). An input (update) u is considered valid and, hence, passes the integrity check iff $\mathsf{Valid}(u) = 1$. For instance, any per-client update check, such as Zeno++ [74], can be a good candidate for $\mathsf{Valid}(\cdot)$ (we evaluate four state-of-the-art validation predicates in Sec. 7.2).

We assume that the honest clients, denoted by C_H: (1) follow the protocol correctly, *and* (2) have well-formed inputs. We require the second condition because, in case the input of an honest client does not pass the integrity check (which can be verified locally since $\mathsf{Valid}(\cdot)$ is public), the client has no incentive to participate in the training iteration.

2.3 Threat Model

We consider a *malicious adversary* threat model:

- **Malicious Server.** We consider a malicious server that can deviate from the protocol arbitrarily with the aim of recovering the raw updates u_i for $i \in [n]$ (see Remark 1 later for more details).
- **Malicious Clients.** We also consider a set of m malicious clients, C_M. Malicious clients can arbitrarily deviate from the protocol with the goals of: (1) sending malformed inputs to the server and thus, compromising the final aggregate; (2) failing the integrity check of an honest client that submits well-formed updates; (3) violating the privacy of an honest client, potentially in collusion with the server.

2.4 Solution Overview

Prior work has mostly focused on ensuring input privacy via secure aggregation, *i.e.*, securely computing the aggregate $\mathcal{U} = \sum_{C_i \in C} u_i$. Motivated by the above problem setting and threat analysis, we introduce a new type of FL protocol, called *secure aggregation with verified inputs* (SAVI), that ensures *both* input privacy and integrity. The goal of a SAVI protocol is to securely aggregate *only* well-informed inputs.

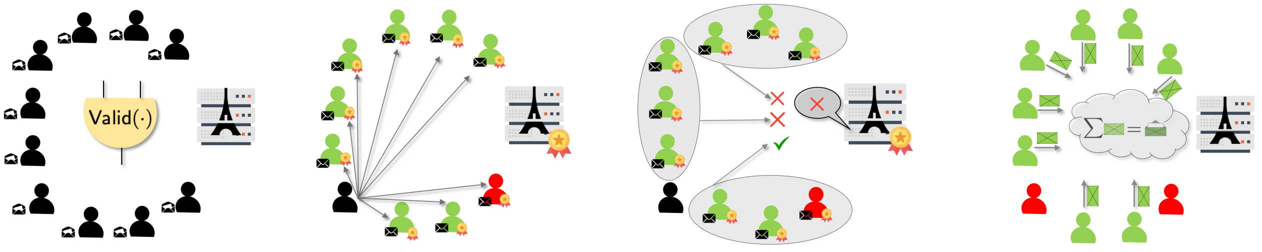
In order to demonstrate the feasibility of SAVI, we propose EIFFeL: a system that instantiates a SAVI protocol for any $\mathsf{Valid}(\cdot)$ that can be expressed as an arithmetic circuit with public parameters (Fig. 1). EIFFeL ensures input privacy by using Shamir's threshold secret sharing scheme [60] (Sec. 4.1). Input integrity is guaranteed via SNIP and verifiable secret shares (VSS) which validates the correctness of the secret shares (Sec. 4.1). The key ideas are:

- SNIP requires multiple honest verifiers. EIFFeL enables this in a single-server setting by having the clients act as the verifiers for each other under the supervision of the server (in Fig. 2b, verifiers are marked by 🏅).
- EIFFeL extends SNIP to the malicious threat model to account for the malicious clients (verifiers). Our key observation is that using a threshold secret sharing scheme creates multiple subsets of clients that can emulate the SNIP verification protocol. The server uses this redundancy to robustly verify the proofs and aggregate updates with verified proofs *only* (Fig. 2c and 2d).

3 Secure Aggregation with Verified Inputs

Below, we provide the formal definition of a *secure aggregation with verified inputs* (SAVI) protocol.

Definition 1. *Given a public validation predicate* $\mathsf{Valid}(\cdot)$ *and security parameter* κ, *a protocol* $\Pi(u_1, \cdots, u_n)$ *is a secure aggregation with verified inputs (SAVI) protocol if:*



(a) EIFFeL consists of multiple clients $\mathcal{C}$ and a server $\mathcal{S}$ with a public validation predicate Valid$(\cdot)$ that defines the integrity check. A client C_i needs to provide a proof π_i for Valid$(u_i) = 1$ (Round 1).

(b) For checking the proof π_i, all other clients $\mathcal{C}_{\setminus i}$ act as the verifiers under the supervision of $\mathcal{S}$. C_i splits its update u_i and proof π_i using Shamir's scheme with threshold $m+1$ and shares it with $\mathcal{C}_{\setminus i}$ (Round 2).

(c) Conceptually, any set of $m+1$ clients in $\mathcal{C}_{\setminus i}$ can emulate the SNIP verification protocol. The server uses this redundancy to *robustly* verify the proof (Round 3).

(d) The clients only aggregate the shares of well-formed updates and the resulting aggregate is revealed to the server (Round 4).

Figure 2: High-level overview of EIFFeL. See Sec. 2.4 for key ideas, and Sec. 4.4 for a detailed description of the system.

- ***Integrity.*** *The output of the protocol, out, returns the aggregate of a subset of clients, $\mathcal{C}_{Valid}$, such that all clients in $\mathcal{C}_{Valid}$ have well-formed inputs.*

$$\Pr\left[out = \mathcal{U}_{Valid}\right] \geq 1 - \text{negl}(\kappa) \text{ where } \mathcal{U}_{Valid} = \sum_{C_i \in \mathcal{C}_{Valid}} u_i$$
$$\text{for all } C_i \in \mathcal{C}_{Valid} \text{ we have } Valid(u_i) = 1$$
$$\mathcal{C}_H \subseteq \mathcal{C}_{Valid} \subseteq \mathcal{C}. \tag{1}$$

- ***Privacy.*** *For a set of malicious clients $\mathcal{C}_M$ and a malicious server $\mathcal{S}$, there exists a probabilistic polynomial-time (P.P.T.) simulator* Sim$(\cdot)$ *such that:*

$$\text{Real}_\Pi(\{u_{\mathcal{C}_H}\}, \Omega_{\mathcal{C}_M \cup \mathcal{S}}) \equiv_C \text{Sim}(\mathcal{U}_H, \mathcal{C}_H, \Omega_{\mathcal{C}_M \cup \mathcal{S}})$$
$$\text{where } \mathcal{U}_H = \sum_{C_i \in \mathcal{C}_H} u_i. \tag{2}$$

$\{u_{\mathcal{C}_H}\}$ denotes the input of all the honest clients, Real$_\Pi$ *denotes a random variable representing the joint view of all the parties in Π's execution, $\Omega_{\mathcal{C}_M \cup \mathcal{S}}$ indicates a polynomial-time algorithm implementing the "next-message" function of the parties in $\mathcal{C}_M \cup \mathcal{S}$ (see full paper [22]), and $\equiv_C$ denotes computational indistinguishability.*

From Def. 1, the output of a SAVI protocol is of the form:

$$\mathcal{U}_{valid} = \underbrace{\mathcal{U}_H}_{\substack{\text{well-formed updates of} \\ \textit{all} \text{ honest clients } \mathcal{C}_H}} + \underbrace{\sum_{C_i \in \mathcal{C}_{\text{Valid}} \setminus \mathcal{C}_H} u_i}_{\substack{\text{well-formed updates of} \\ \text{some malicious clients}}}. \tag{3}$$

The clients in $\mathcal{C}_{\text{Valid}} \setminus \mathcal{C}_H$ are clients who have submitted well-formed inputs but can behave maliciously otherwise (*e.g.*, by violating input privacy/integrity of honest clients).

The privacy constraint of the SAVI protocol means that a simulator Sim can generate the views of all parties with just access to the list of the honest clients $\mathcal{C}_H$ and their aggregate $\mathcal{U}_H$. Note that Sim takes $\mathcal{U}_H$ as an input instead of the protocol output $\mathcal{U}_{\text{Valid}}$. This is because the clients in $\mathcal{C}_{\text{Valid}} \setminus \mathcal{C}_H$, by virtue of being malicious, can behave arbitrarily and announce their updates to reveal $\mathcal{U}_H = \mathcal{U}_{\text{Valid}} - \sum_{C_i \in \mathcal{C}_{\text{Valid}} \setminus \mathcal{C}_H} u_i$. Thus, SAVI ensures that nothing can be learned about the input u_i of an honest client $C_i \in \mathcal{C}_H$ except:

- that u_i is well-formed, *i.e.*, Valid$(u_i) = 1$,
- anything that can be learned from the aggregate $\mathcal{U}_H$.

Remark 1. Note that we consider a malicious server only for input privacy and the reason is as follows. For input integrity, a malicious server can do the following:

- **Case 1.** Mark the input of an honest client as invalid and not include it in the final aggregate.
- **Case 2.** Mark the (invalid) input of a malicious client as valid.

EIFFeL prevents Case 1 from happening since, from Def. 1, EIFFeL is guaranteed to output the aggregate of *all* honest clients (Lemma 4). If we consider a malicious server even for data integrity, the only thing that can happen is Case 2. However, the server's primary goal is to ensure that each input is well-formed. Hence, Case 2, i.e., marking the (invalid) input of a malicious client as valid, is at odds with the server's goal. Therefore, it is unnecessary to protect against this behavior in our setting and we consider the server to be honest for the purposes of input integrity.

Remark 2. The integrity constraint of SAVI requires the protocol to detect *and* remove *all* malformed inputs before computing the final aggregate. Note that there is a fundamental difference between the design choice of just detection of a malformed input versus detection *and* removal. In the former, the server can only abort the current round even when just a *single* malformed input is detected. This allows an adversary to stage a denial-of-service attack that renders the server incapable of training the model. When the protocol can both detect and remove malformed inputs, such denial-of-service attacks are prevented as the server can train the model using just the valid updates.

4 EIFFeL System Description

This section introduces EIFFeL: the system we propose to perform secure aggregation of verified inputs.

4.1 Cryptographic Building Blocks

Arithmetic Circuit. An arithmetic circuit, $C : \mathbb{F}^k \mapsto \mathbb{F}$, represents a computation over a finite field $\mathbb{F}$. Conceptually, it is similar to a Boolean circuit but it uses finite field addition, multiplication and multiplication-by-constant instead of OR, AND, and NOT gates.

Shamir's t-out-of-n Secret Sharing Scheme [60] allows distributing a secret s among n parties such that: (1) the complete secret can be reconstructed from any combination of t shares; (2) any set of $t-1$ or fewer shares reveals no information about s where t is the *threshold* of the secret sharing scheme. The scheme is parameterized over a finite field $\mathbb{F}$ and consists of two algorithms:

- $\{(i, s_i)\}_{i \in P} \xleftarrow{\$} \mathsf{SS.share}(s, P, t)$. Given a secret $s \in \mathbb{F}$, a set of n unique field elements $P \in \mathbb{F}^n$ and a threshold t with $t \leq n$, this algorithm constructs n shares. The algorithm chooses a random polynomial $p \in \mathbb{F}[X]$ such that $p(0) = s$ and generates the shares as $(i, p(i)), i \in P$.
- $s \leftarrow \mathsf{SS.recon}(\{(i, s_i)_{i \in Q}\})$. Given the shares corresponding to a subset $Q \subseteq P, |Q| \geq t$, the reconstruction algorithm recovers the secret s.

Shamir's secret sharing scheme is *linear*, which means a party can *locally* perform: (1) addition of two shares, (2) addition of a constant, and (3) multiplication by a constant.
Shamir's secret sharing scheme is closely related to Reed-Solomon error correcting codes [45], which is a group of polynomial-based error correcting codes. The share generation is similar to (non-systemic) message encoding in these codes which can successfully recover a message even in the presence of errors and erasures (message dropouts). Consequently, we can leverage Reed-Solomon decoding for robust reconstruction of Shamir's secret shares:

- $s \leftarrow \mathsf{SS.robustRecon}(\{(i, s_i)\}_{i \in Q})$. Shamir's secret sharing scheme results in a $[n, t, n-t+1]$ Reed-Solomon code that can tolerate up to q errors and e erasures (message dropouts) such that $2q + e < n - t + 1$. Given any subset of n–e shares $Q \subseteq P, |Q| \geq n - e$ with up to q errors, any standard Reed Solomon decoding algorithm [13] can robustly reconstruct s. EIFFeL uses Gao's decoding algorithm [32].

Verifiable secret sharing scheme is a related concept where the scheme has an additional property of *verifiability.* Given a share of the secret, a party must be able to check whether it is indeed a valid share. If a share is valid, then there exists a unique secret which will be the output of the reconstruction algorithm when run on any t distinct valid shares. Formally:

- $1/0 \leftarrow \mathsf{SS.verify}((i, v), \Psi))$. The verification algorithm inputs a share and a check string Ψ_s such that

$$\forall\, V \subset \mathbb{F} \times \mathbb{F} \text{ where } |V| = t, \exists s \in \mathbb{F} \text{ s.t.}$$
$$(\forall (i, v) \in V, \mathsf{SS.verify}((i, v), \Psi_s) = 1) \implies \mathsf{SS.recon}(V) = s$$

  The share construction algorithm is augmented to output the check string as $(\{(i, s_i)_{i \in P}\}, \Psi_s) \leftarrow \mathsf{SS.share}(s, P, t)$.

For EIFFeL, we use the non-interactive verification scheme by Feldman [30] (details in the full paper [22]).

Key Agreement Protocol. A key agreement protocol consists of a tuple of the following three algorithms:

- $(pp) \xleftarrow{\$} \mathsf{KA.param}(1^{\kappa})$. The parameter generation algorithm samples a set of public parameters pp with security parameter κ.
- $(pk, sk) \xleftarrow{\$} \mathsf{KA.gen}(pp)$. The key generation algorithm samples a public/secret key pair from the public parameters.
- $sk_{ij} \leftarrow \mathsf{KA.agree}(pk_i, sk_j)$. The key agreement protocol receives a public key pk_i and a secret key sk_j as input and generates the shared key sk_{ij}.

Authenticated Encryption provides confidentiality and integrity guarantees for messages exchanged between two parties. It consists of a tuple of three algorithms as follows:

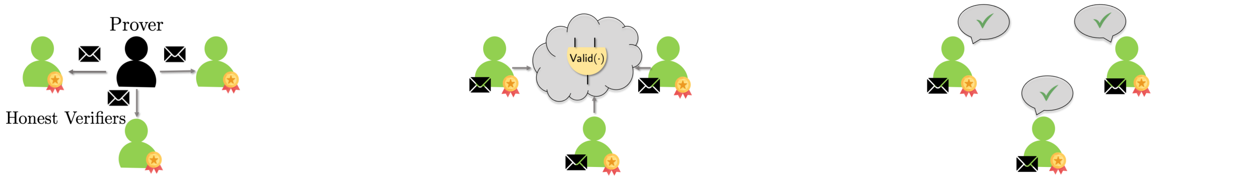
- $k \xleftarrow{\$} \mathsf{AE.gen}(1^{\kappa})$. The key generation algorithm that outputs a private key k where κ is the security parameter.
- $\overline{x} \xleftarrow{\$} \mathsf{AE.enc}(k, x)$. The encryption algorithm takes as input a key k and a message x, and outputs a ciphertext $\overline{x}$.
- $x \leftarrow \mathsf{AE.dec}(k, \overline{x})$. The decryption algorithm takes as input a ciphertext and a key and outputs either the original plaintext, or a special error symbol $\perp$ on failure.

Secret-shared Non-interactive Proofs. The secret-shared non-interactive proof (SNIP) [24] is an information-theoretic zero-knowledge proof for distributed data (Fig. 3). SNIP is designed for a multi-verifier setting where the private data is distributed or secret-shared among the verifiers. Specifically, SNIP relies on an additive secret sharing scheme over a field $\mathbb{F}$ as described below. A secret $s \in \mathbb{F}$ is split into k random shares $([s]_1, \cdots, [s]_k)$ such that $\sum_{i=1}^{k} [s]_i = s$. A subset of up to $k-1$ shares reveals *no* information about the secret s. The additive secret-sharing scheme is linear as well.

SNIP Setting. SNIP considers $k \geq 2$ verifiers $\{\mathcal{V}_i\}, i \in [k]$ and a prover $\mathcal{P}$ with a private vector $x \in \mathbb{F}^d$. All parties also hold a *public* arithmetic circuit representing a validation predicate $\mathsf{Valid} : \mathbb{F}^d \mapsto \mathbb{F}$. Let M be the number of multiplication gates in $\mathsf{Valid}(\cdot)$. $\mathbb{F}$ is chosen such that $2\mathsf{M} \ll |\mathbb{F}|$. The prover $\mathcal{P}$ splits x into k shares $\{[x_1], \cdots, [x_k]\}$. Next, they generate k proof strings $[\pi]_i, i \in [k]$ based on $\mathsf{Valid}(\cdot)$ and shares $([x_i], [\pi]_i)$ with every verifier $\mathcal{V}_i$ (Fig. 3a).
The prover's goal is to convince the verifiers that, indeed, $\mathsf{Valid}(x) = 1$. The prover does so via proof strings $[\pi]_i, i \in [k]$, that do not reveal anything else about x. After receiving the proof, the verifiers gossip with each other to conclude either that $\mathsf{Valid}(x) = 1$ (the verifiers "Accept x") or not ("Reject x", Figs. 3b and 3c). Formally, SNIP satisfies the following security properties:

- *Completeness.* If all parties are honest and $\mathsf{Valid}(x) = 1$, then the verifiers will accept x.
$$\forall x \in \mathbb{F} \text{ s.t. } \mathsf{Valid}(x) = 1 : \Pr_{\pi}[\mathsf{Accept}\ x] = 1.$$
- *Soundness.* If all verifiers are honest, and if $\mathsf{Valid}(x) = 0$, then for all malicious provers, the verifiers will reject x with overwhelming probability.
$$\forall x \in \mathbb{F} \text{ s.t. } \mathsf{Valid}(x) = 0 : \ \Pr_{\pi}\big[\mathsf{Reject}\ x\big] \geq 1 - {}^{(2M-2)}/{|\mathbb{F}|}.$$
- *Zero knowledge.* If the prover and at least one verifier are honest, then the verifiers learn nothing about x, except that $\mathsf{Valid}(x) = 1$. Formally, when $\mathsf{Valid}(x) = 1$, there exists a simulator $\mathsf{Sim}(\cdot)$ that can simulate the view of the protocol execution for every proper subset of verifiers:
$$\forall x \text{ s.t. } \mathsf{Valid}(x) = 1 \text{ and } \forall \bar{\mathcal{V}} \subset \bigcup_{i=1}^{k} \mathcal{V}_i \text{ we have}$$
$$\mathrm{Sim}_{\pi}(\mathsf{Valid}(\cdot), \{([x]_i, [\pi]_i)\}_{i \in \bar{\mathcal{V}}}) \equiv \mathrm{View}_{\pi, \bar{\mathcal{V}}}(\mathsf{Valid}(\cdot), x).$$

Thus, SNIP allows the verifiers to collaboratively check – without ever accessing the prover's private data in the clear – that the



(a) Prover sends secret shares of its input and the SNIP proof to multiple verifiers.

(b) The verifiers gossip among themselves and check the proof.

(c) The check passes successfully if all verifiers are honest.

Figure 3: High-level overview of a secret-shared non-interactive proof (SNIP; [24]).

prover's submission is, indeed, well-formed. SNIP works in two stages as follows:

(1) *Generation of Proof.* For generating the proof, the prover $\mathcal{P}$ first evaluates the circuit $\mathsf{Valid}(\cdot)$ on its input x to obtain the value of every wire in the arithmetic circuit corresponding to the computation of $\mathsf{Valid}(x)$. Using these wire values, $\mathcal{P}$ constructs three polynomials f, g, and h of the lowest possible degrees such that $h = f \cdot g$ and $f(j), g(j)$ and $h(j), j \in [\mathsf{M}]$ encode the values of the two input wires and one output wire of the j-th multiplication gate, respectively. $\mathcal{P}$ also samples a single set of Beaver's multiplication triples [7]: $(a, b, c) \in \mathbb{F}^3$ such that $a \cdot b = c \in \mathbb{F}$. Finally, it generates the shares of the proof, $[\pi]_i = ([h]_i, ([a]_i, [b]_i, [c]_i))$, which consists of:

- shares of the coefficients of the polynomial h, denoted by $[h]_i$,
- shares of the Beaver's triples, $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$.

The prover then sends the respective shares of the input and the proof $([x]_i, [\pi]_i)$ to each of the verifiers $\mathcal{V}_i$.

(2) *Verification of Proof.* To verify that $\mathsf{Valid}(x) = 1$ and hence, accept the input x, the verifiers need to check two things:

- check that the value of final output wire of the computation, $\mathsf{Valid}(x)$, denoted by w^{out} is indeed 1, and
- check the consistency of $\mathcal{P}$'s computation of $\mathsf{Valid}(x)$.

To this end, each verifier $\mathcal{V}_i$ *locally* constructs the shares of every wire in $\mathsf{Valid}(x)$ via affine operations on the shares of the private input $[x]_i$ and $[h]_i$. Next, $\mathcal{V}_i$ broadcasts a summary $[\sigma]_i = ([w^{out}]_i, [\lambda]_i)$, where $[w^{out}]_i$ is $\mathcal{V}_i$'s share of the output wire of the circuit and $[\lambda]_i$ is a share of a random digest that the verifier computes from the shares of the other wire values and the proof string $[\pi]_i$. Using these summaries, the verifiers check the proof as follows:

- For checking the output wire, the verifiers can reconstruct its exact value from all the broadcasted shares $w^{out} = \sum_{i=1}^{k}[w^{out}]_i$ and check whether $w^{out} = 1$. This would imply that $\mathsf{Valid}(x) = 1$.
- The circuit consistency check is more involved and is performed using the random digest λ. First, $\mathcal{V}_i$ *locally* computes the shares of the polynomials f and g (denoted as $[f]_i$ and $[g]_i$). To verify the consistency of the circuit evaluation, the verifiers need to check that the shares $[h]_i$ sent by the prover $\mathcal{P}$ are of the correct polynomial, *i.e.*, confirm that $f \cdot g = h$. For this, SNIP uses the Schwartz-Zippel polynomial identity test [59, 79]. Specifically, verifiers reconstruct $\lambda = \sum_{1=1}^{k}[\lambda]_i$ from the broadcasted shares and test whether $\lambda = r(f(r) \cdot g(r) - h(r)) = 0$ on a randomly selected $r \in \mathbb{F}$. The computation of the share of the random digest $[\lambda]_i$ uses the shares of Beaver's triples $([a]_i, [b]_i, [c]_i)$.

A detailed description SNIP is in the full paper [22].

4.2 System Building Blocks

Public Validation Predicate. EIFFeL requires a public validation predicate $\mathsf{Valid}(\cdot)$, expressed by an arithmetic circuit, that captures the notion of update well-formedness. In principle, any per-client update robustness test [5, 27, 44, 61, 67, 68, 74] from the ML literature can be a suitable candidate. The parameters of the test (for instance, threshold ρ for a norm bound check $\mathsf{Valid}(u) = \mathbb{I}[\|u\|_2 < \rho)$ can be computed from a clean, public dataset $\mathcal{D}_P$ that is available to the server $\mathcal{S}$. This assumption of a clean, public dataset is common in both ML [19, 36, 74] as well as privacy literature [6, 8, 47]. The dataset can be small and obtained by manual labeling [49].

Public Bulletin Board. EIFFeL assumes the availability of a public bulletin board $\mathcal{B}$ that is accessible to all the parties, similar to prior work [15, 36, 58]. In practice, the bulletin $\mathcal{B}$ can be implemented as an append-only log hosted at a public web address where every message and its sender is visible. Every party in EIFFeL has read/write access to it. We use the bulletin $\mathcal{B}$ as a tool for broadcasting [17, 26].

4.3 EIFFeL Design Goals

In terms of the design, EIFFeL should:

- provide *flexibility in the choice of integrity checks.*
- be *compatible with the existing FL infrastructure in deployment.*
- be *efficient* in performance.

4.4 EIFFeL Workflow

The goal of EIFFeL is to instantiate a secure aggregation with verified inputs (SAVI) protocol in FL. For a given public validation predicate $\mathsf{Valid}(\cdot)$, EIFFeL checks the integrity of every client update using SNIP and outputs the aggregate of *only* well-formed updates, *i.e.*, $\mathsf{Valid}(u) = 1$. To implement SNIP for our setting, EIFFeL introduces two main ideas:

Main Ideas.

- In EIFFeL, the clients act as the verifiers for each other. Specifically, for every client $C_i, i \in C$, all of the other $n-1$ clients, $C_{\setminus i}$, and the server $\mathcal{S}$ jointly acts as the verifiers. This is different from Prio [24] (the original SNIP deployment setting) that rely on multiple *honest servers* to perform verification.
- In EIFFeL, verification can be performed even with malicious verifiers. This is essential in our setting since we have m malicious clients (*i.e.*, verifiers).[a] For this, EIFFeL uses Shamir's t-out-of-n threshold scheme for the entire protocol. This allows any cohort of t verifiers to reconstruct a secret and, hence, instantiate a SNIP protocol. If $t < n$, we have multiple such instantiations and can use the redundancy to perform the integrity check even with some of the verifiers being malicious.

[a] The server works honestly towards verifying the integrity (its primary goal) but could behave maliciously (possibly colluding with other malicious clients) to violate the privacy of the honest clients (see Sec. 2.2).

The full protocol is presented in Fig. 4. The protocol involves a setup phase followed by four rounds.

Setup Phase. In the setup phase, all parties are initialized with the system-wide parameters, namely the security parameter κ, the number of clients n out of which *only* $m < \lfloor \frac{n-1}{3} \rfloor$ can be malicious, public parameters for the key agreement protocol $pp \overset{\$}{\leftarrow} \mathsf{KA.param}(\kappa)$, and a field $\mathbb{F}$ where $|\mathbb{F}| \geq 2^{\kappa}$. EIFFeL works in a synchronous protocol between the server $\mathcal{S}$ and the n clients in four rounds. To prevent the server from simulating an arbitrary number of clients, the clients register themselves with a specific user ID on the public bulletin board $\mathcal{B}$ and are authenticated with the help of standard public key infrastructure (PKI). The bulletin board $\mathcal{B}$ allows parties to register IDs only for themselves, preventing impersonation. More concretely, the PKI enables the clients to register identities (public keys), and sign messages using their identity (associated secret keys), such that others can verify this signature, but cannot impersonate them [37]. We omit this detail for the ease of exposition. For notational simplicity, we assume that each client C_i is assigned a unique logical ID in the form of an integer i in $[n]$. Each client holds as input a d-dimensional vector $u_i \in \mathbb{F}^d$ representing its local update. All clients have a private, authenticated communication channel with the server $\mathcal{S}$. Additionally, every party (clients and server) has read and write access to the public bulletin $\mathcal{B}$ via authenticated channels. For every client C_i, the server $\mathcal{S}$ maintains a list, $\mathsf{Flag}[i]$, of all clients that have flagged C_i as malicious. All $\mathsf{Flag}[i]$ lists are initialized to be empty lists.

Round 1 (Announcing Public Information). In the first round, all the parties announce their public information relevant to the protocol on the public bulletin $\mathcal{B}$. Specifically, each client C_i generates its key pair $(pk_i, sk_i) \overset{\$}{\leftarrow} \mathsf{KA.gen}(pp)$ and advertises the public key pk_i on the public bulletin $\mathcal{B}$. The server $\mathcal{S}$ publishes the validation predicate $\mathsf{Valid}(\cdot)$ on $\mathcal{B}$.

Round 2 (Generate and Distribute Proofs). Every client generates shares of its private update u_i and the proof π_i, and distributes these shares to the other clients $C_{\setminus i}$. First, client C_i generates a common pairwise encryption key sk_{ij} for every other client $C_j \in C_{\setminus i}$ using the key agreement protocol, $sk_{ij} \leftarrow \mathsf{KA.agree}(sk_i, pk_j)$. Next, the client generates the secret shares of its private update $\{(1, u_{i1}), \cdots, (n, u_{in}), \Psi_{u_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(u, [n], m+1)$. The sharing of u_i is performed dimension-wise; we abuse notations and denote the j-th such share by $(j, u_{ij}), j \in [n]$. Note that the client C_i generates a share (i, u_{ii}) for *itself* as well which will be used later in the protocol. Next, the client C_i generates the proof for the computation $\mathsf{Valid}(u_i) = 1$. Specifically, it computes the polynomials f_i, g_i, and $h_i = f_i \cdot g_i$ and samples a set of Beaver's multiplication triples $(a_i, b_i, c_i) \in \mathbb{F}^3, a_i \cdot b_i = c_i \in \mathbb{F}$. Since the other clients will verify the proof, client C_i then splits the proof to generate shares $\pi_{ij} = \big((j, h_{ij}), (j, a_{ij}), (j, b_{ij}), (j, c_{ij})\big)$ for every other client $C_j \in C_{\setminus i}$. The shares themselves are generated via $\{(1, h_{i1}), \cdots, (i-1, h_{i(i-1)}), (i+1, h_{i(i+1)}), \cdots, (n, h_{in}), \Psi_{h_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(h_i, [n] \setminus i, m+1)$, and so on. Finally, the client encrypts the proof strings (shares of the update u_i and the proof π_i) using the corresponding pairwise secret key, $\overline{(j, u_{ij}) || (j, \pi_{ij})} \overset{\$}{\leftarrow} \mathsf{AE.enc}\big(sk_{ij}, (j, u_{ij}) || (j, \pi_{ij})\big)$, and publishes the encrypted proof strings on the public bulletin $\mathcal{B}$. The client also publishes the check strings Ψ_{u_i} and $\Psi_{\pi_i} = (\Psi_{h_i}, \Psi_{a_i}, \Psi_{b_i}, \Psi_{c_i})$ for verifying the validity of the shares of u_i and π_i, respectively.

Round 3 (Verify Proof). In this round, every client C_i partakes in the verification of the proofs π_j of all other clients $C_j \in C_{\setminus i}$, under the supervision of the server $\mathcal{S}$. The goal of the server is to identify the malicious clients, C_M. To this end, the server maintains a (partial) list, C^* (initialized as an empty list), of clients it has so far identified as malicious. The proof-verification round consists of three phases as follows:

(i) *Verifying the validity of the secret shares.* First, every client C_i downloads and decrypts their shares from the bulletin $\mathcal{B}$, $(i, u_{ji}) || (i, \pi_{ji}) \leftarrow \mathsf{AE.dec}\big(sk_{ij}, \overline{(i, u_{ji}) || (i, \pi_{ji})}\big), \forall C_j \in C_{\setminus i}$. Additionally, C_i downloads the check strings $(\Psi_{u_i}, \Psi_{\pi_i})$ and verifies the validity of the shares. If the shares from any client C_j:

- fail to be decrypted, *i.e.*, $\mathsf{AE.dec}(\cdot)$ outputs $\perp$, OR
- fail to pass the verification, *i.e.*, $\mathsf{SS.verify}(\cdot)$ returns 0,

C_i flags C_j on the bulletin $\mathcal{B}$. Every time a client C_i flags another client C_j, the server updates the corresponding list $\mathsf{Flag}[j] \leftarrow \mathsf{Flag}[j] \cup C_i$. If $|\mathsf{Flag}[j]| \geq m+1$, the server $\mathcal{S}$ marks C_j as malicious: $C^* \leftarrow C^* \cup C_j$. The server can do so because the pigeon hole principle implies that C_j must have sent an invalid share to at least one honest client; hence, the correctness of the value recovered from that client's shares cannot be guaranteed. In case $1 \leq |\mathsf{Flag}[j]| \leq m$, the server supervises the following actions. Suppose client C_i has flagged client C_j. Client C_j then reveals the shares for C_i, $\big((i, u_{ji}), (i, \pi_{ji})\big)$ in the clear (on bulletin $\mathcal{B}$) for the server $\mathcal{S}$ (or anyone else) to verify using $\mathsf{SS.Verify}(\cdot)$. If that verification passes, C_i is instructed by the server to use the released shares for its computations. Otherwise, C_j is marked as malicious by the server $\mathcal{S}$. Note that this does not lead to privacy violation for an honest client since at most m shares corresponding to the m malicious clients would be revealed (see Sec. 5). If a client C_i flags $\geq m+1$ other clients, $\mathcal{S}$ marks C_i as malicious. Thus, at this point every client on the list C^* has either

- provided invalid shares to at least one honest client, OR
- flagged an honest client.

In other words, every client who is *not* in C^*, $C_i \in C \setminus C^*$, is guaranteed to have submitted at least $n-m-1$ valid shares for the honest clients in $C_H \setminus C_i$ (see Sec. 5 for details). Additionally, the server cannot be tricked into marking an honest client as malicious, *i.e.*, EIFFeL ensures $C^* \cap C_H = \varnothing$ (see Sec. 5). The server $\mathcal{S}$ publishes C^* on the bulletin $\mathcal{B}$.

(ii) *Computation of proof summaries by clients.* For this phase, the server $\mathcal{S}$ advertises a random value $r \in \mathbb{F}$ on the bulletin $\mathcal{B}$. Next, a client C_i proceeds to distill the proof strings of all clients *not* in C^* to generate summaries for the server $\mathcal{S}$. Specifically, client C_i prepares a proof summary $\sigma_{ji} = \big((i, w_{ji}^{out}), (i, \lambda_{ji})\big)$ for $\forall C_j \in C \setminus (C^* \cup C_i)$ as per the description in the previous section, and publishes it on $\mathcal{B}$.

(iii) *Verification of proof summaries by the server.* Next, the server moves to the last step of verifying the proof summaries $\sigma_i = (w_i^{out}, \lambda_i)$ for all clients not in C^*. Recall from the discussion in Sec. 4.1 that this involves recovering the values w_i^{out} and λ_i from the shares of σ_i and checking whether $w_i^{out} = 1$ and $\lambda_i = 0$. However, we cannot simply use the naive share reconstruction algorithm from Sec. 4.1 since some of the shares might be incorrect (submitted by the malicious clients). To address this issue, EIFFeL performs a *robust reconstruction* of the shares as follows. A naive strategy would be sampling multiple subsets of $m + 1$ shares (each subset can emulate a SNIP setting), reconstructing the secret for each subset, and taking the majority vote. However, we can do much better by exploiting the connections between Shamir's secret shares and Reed-Solomon error correcting codes (Sec. 4.1). Specifically, the Shamir's secret sharing scheme used by EIFFeL is a $[n-1, m+1, n-m]$ Reed-Solomon code that can correct up to q errors and e erasures (message dropouts) where $2q + e < n-m-1$. The server $\mathcal{S}$ can, therefore, use $\text{SS.robustRecon}(\cdot)$ to reconstruct the secret when $m < \lfloor \frac{n-1}{3} \rfloor$.

After the robust reconstruction of the proof summaries, the server $\mathcal{S}$ verifies them and updates the list $\mathcal{C}^*$ with *all* malicious clients with malformed updates. Specifically:

$$\forall C_i \in \mathcal{C} \setminus \mathcal{C}^*$$

$$\Big(\text{SS.robustRecon}(\{(j, w_{ij}^{out})\}_{C_j \in \mathcal{C} \setminus \{\mathcal{C}^* \cup C_i\}}) \neq 1 \vee$$

$$\text{SS.robustRecon}(\{(j, \lambda_{ij})\}_{C_j \in \mathcal{C} \setminus \{\mathcal{C}^* \cup C_i\}}) \neq 0\Big)$$

$$\implies \mathcal{C}^* \leftarrow \mathcal{C}^* \cup C_i.$$

Additionally, if a client C_i withholds some of the shares of the proof summaries for other clients, C_i is marked as malicious as well by the server. Thus, in addition to the malicious clients listed above, the list $\mathcal{C}^*$ now has all clients that have either:

- failed the proof verification, *i.e.*, provided malformed updates, OR
- withheld shares of proof summaries of other clients (malicious message dropout).

To conclude the round, the server publishes the updated list $\mathcal{C}^*$ on the public bulletin $\mathcal{B}$.

Round 4 (Compute Aggregate). This is the final round of EIFFeL where the aggregate of the well-formed updates is computed. If a client C_i is on $\mathcal{C}^*$ wrongfully, it can dispute its malicious status by showing the other clients the transcript of the robust reconstruction from all the shares of σ_i (publicly available on bulletin $\mathcal{B}$). If any client $C_i \in \mathcal{C}$ successfully raises a dispute, all clients abort the protocol because they conclude that the server $\mathcal{S}$ has acted maliciously by trying to withhold a verified well-formed update from the aggregation. If no client raises a successful dispute, every client $C_i \in \mathcal{C} \setminus \mathcal{C}^*$ generates its share of the aggregate, $(i, \mathcal{U}_i)$ with $\mathcal{U}_i = \sum_{C_j \in \mathcal{C} \setminus \mathcal{C}^*} u_{ji}$, and sends that share to the server $\mathcal{S}$. Note that, herein, C_i uses its own share of the update, (i, u_{ii}), as well.

The server recovers the aggregate $\mathcal{U} = \sum_{C_i \in \mathcal{C} \setminus \mathcal{C}^*} \mathcal{U}_j$ using robust reconstruction: $\mathcal{U} \leftarrow \text{SS.robustRecon}(\{(i, \mathcal{U}_i)\}_{C_i \in \mathcal{C} \setminus \mathcal{C}^*})$.

Discussion. EIFFeL meets the design goals of Sec. 4.3 as follows.

Flexibility of Integrity Checks. SNIP supports arbitrary arithmetic circuits for $\text{Valid}(\cdot)$. The server $\mathcal{S}$ can choose a different $\text{Valid}(\cdot)$ for every iteration (the protocol described above corresponds to a single iteration of model training in FL). Additionally, $\mathcal{S}$ can hold multiple $\text{Valid}_1(\cdot), \cdots, \text{Valid}_k(\cdot)$ and want to check whether the client's update passes them all. For this, we have $\text{Valid}_i(\cdot)$ return zero (instead of one) on success. If w_i^{out} is the value on the output wire of the circuit $\text{Valid}_i(\cdot)$, the server chooses random values $(l_1, \cdots, l_k) \in \mathbb{F}^k$ and recovers the sum $\sum_{i=1}^{k} l_i \cdot w_i^{out}$ in Round 3. If any $w_i^{out} = 0$, then the sum will be non-zero with high probability and $\mathcal{S}$ will reject.

Compatibility with FL's Infrastructure. Current deployments of FL involves a *single* server who wants to train the global model. Hence, as explained above, we design SNIP to be compatible with a single server in EIFFeL. Solutions involving two or more non-colluding servers are unrealistic for FL. For instance, currently the server can be owned by Meta who wants to train privately on the data of its user base. For a two-server model here, the second server has to be owned by an independent party. Moreover, both the servers have to do an equal amount of computation for model training (verification, aggregation etc) since SNIP uses secret shares. This would make sense only if *both* the servers are interested in training the model. For instance, if Meta and Google collaborate to train a model on their joint user base which is an unrealistic scenario.

Efficiency. EIFFeL's usage of SNIP as the underlying ZKP is made from the efficiency point of view. SNIP is a light-weight ZKP system that is *specialized for the server-client settings* resulting in good performance. For instance, its performance is about three-orders of magnitude better than that of zkSNARKs [24]. Instead of using ZKPs, one alternative is to use standard secure multi-party computation (MPC) for the entire aggregation to directly compute $\mathcal{U}_{valid} = \sum_{C_i} \text{Valid}(u_i) \cdot u_i$. However, doing the entire aggregation under MPC would result in a massive circuit with $O(nd)$ multiplication gates where d is the data dimension. Multiplications are costly for MPC and each gate requires a round of communication in general making the above computation prohibitively costly. Extending the computation to the malicious threat model would be even costlier. This is where SNIP proves to be advantageous: SNIP enables the verifiers to check all the multiplication gates very efficiently (in a non-interactive fashion) with just one polynomial identity test (Sec. 4.1).

Remark 3. In a nutshell, EIFFeL's technical novelty is in providing an *efficient* extension of SNIP to a (1) fully malicious threat model in a (2) single server setting.

EIFFeL's problem setting is different from that of the original SNIP proposal in the following ways:

- The original SNIP deployment setting (Prio) uses ≥ 2 non-colluding servers as the verifiers; EIFFeL requires a single server.
- Originally, SNIP considers honest verifiers; EIFFeL supports a fully malicious threat model (provers and verifiers).

The above changes cannot be supported in SNIP as is and are necessary to capture the constraints of a realistic FL setting.

The core essence of the technical differences between the original deployment of SNIP and EIFFeL is that the roles of the parties are changed: the former has a clear distinction between the prover (clients) and verifiers ($k \geq 2$ honest servers), whereas the clients and the single server jointly act as the (malicious) verifiers in EIFFeL.

Consequently, SNIP's interaction pattern is different in EIFFeL which required the following technical changes.

- Originally, SNIP uses additive secret shares whereas EIFFeL uses Shamir's threshold secret sharing.

	Computation	Communication
Client	$O(mnd)$	$O(mnd)$
Server	$O\big((n+d)n\log^2 n \log\log n + md\min(n,m^2)\big)$	$O\big(n^2 + md\min(n,m^2)\big)$

Table 1: Computational and communication complexity of EIFFeL for the server and an individual client.

- In the original construction of SNIP, the reconstruction (a fundamental operation in the protocol) of shares is very simple: just add (+) the individual shares. In EIFFeL, we propose a robust reconstruction technique, SS.robustRecon$(\cdot)$, based on Reed-Solomon codes. This is key in ensuring robust verification even with malicious verifiers.
- EIFFeL, in addition, requires verifiable secret shares: shares are augmented with a check string Ψ which is integrated into the protocol.
- Messages (shares, proofs) are distributed in the clear in the original SNIP deployment; messages are encrypted in EIFFeL.
- We propose novel optimizations for EIFFeL (Sec. 6.1 targets the new operations for verifying secret shares and robust reconstruction while Sec. 6.2 provides general improvements for SNIP).

Table 1 analyses the complexity of EIFFeL in terms of the number of clients n, number of malicious clients m and data dimension d. We assume that $|\mathsf{Valid}|$ is of the order of $O(d)$. The total number of one-way communication is 12 and 9 for the clients and the server, respectively. A per-round analysis is presented in the full paper [22].

5 Security Analysis

In this section, we formally analyze the security of EIFFeL.

Theorem 1. *For any public validation predicate* $\mathit{Valid}(\cdot)$ *that can be expressed by an arithmetic circuit, EIFFeL is a SAVI protocol (Def. 1) for* $|C_M| < \lfloor \frac{n-1}{3} \rfloor$ *and* $C_{\mathit{Valid}} = C \setminus C^*$.

We present a proof sketch of the above theorem here; the formal proof is in the full paper [22].

Proof Sketch. The proof relies on the following two facts.

Fact 1. *Any set of* m *or less shares in EIFFeL reveals nothing about the secret.*

Fact 2. *A* $(n, m+1, n-m)$ *Reed-Solomon error correcting code can correctly construct the message with up to* q *errors and* e *erasures (message dropout), where* $2q+e < n-m+1$. *In EIFFeL, we have* $q+e=m$ *where* q *is the number of malicious clients that provide erroneous shares and* e *is the number of clients that withhold a message or are barred from participation (*i.e.*, are in* C^**).*

Integrity. We prove that EIFFeL satisfies the integrity constraint of the SAVI protocol using the following three lemmas.

Lemma 2. *EIFFeL accepts the update of every honest client.*

$$\forall C_i \in C_H : \Pr_{\mathit{EIFFeL}}[\mathit{Accept}\, u_i] = 1. \tag{4}$$

Proof. By definition, client $C_i \in C_H$ has well-formed inputs, that is, $\mathsf{Valid}(u_i) = 1$. Additionally, C_i, by virtue of being honest, submits valid shares. Hence, at least $n-m-1$ other honest clients $C_H \setminus C_i$ will produce correct shares of the proof summary $\sigma_i = (w_i^{out}, \lambda_i)$. Using Fact 2, the server $\mathcal{S}$ is able to correctly reconstruct the value of σ_i. Eq. 4 is now implied by the completeness property of SNIP. □

Lemma 3. *All updates accepted by EIFFeL are well-formed with probability* $1 - \mathrm{negl}(\kappa)$.

$$\forall C_i \in C, \Pr_{\mathit{EIFFeL}}\big[\mathit{Valid}(u_i) = 1 \big| \mathit{Accept}\, u_i\big] = 1 - \mathrm{negl}(\kappa). \tag{5}$$

The proof relies on the fact that a client will be verified only if it has submitted $\geq n-m-1$ valid shares (see full paper [22]).

Corollary 3.1. *EIFFeL rejects all malformed updates with probability* $1 - \mathrm{negl}(\kappa)$.

Based on the above lemmas, at the end of Round 3, $C \setminus C^*$ (set of clients whose updates have been accepted) must contain *all* honest clients C_H. Additionally, it may contain some clients C_i who have submitted well-formed updates with at least $n-m-1$ valid shares for C_H, but may act maliciously for other steps of the protocol (for instance, give incorrect shares of proof summary for other clients or give incorrect shares of the final aggregate). This is acceptable provided that EIFFeL is able to reconstruct the final aggregate containing *only* well-formed updates which is guaranteed by the following lemma.

Lemma 4. *The aggregate* $\mathcal{U}$ *must contain the updates of all honest clients or the protocol is aborted.*

$$\mathcal{U} = \mathcal{U}_H + \sum_{C_i \in \bar{C}} u_i \text{ where } \mathcal{U}_H = \sum_{C_i \in C_H} u_i$$
$$\bar{C} \subseteq C \setminus \{C^* \cup C_H\} \tag{6}$$

Proof. If the server $\mathcal{S}$ acts maliciously and publishes a list C^* such that $C^* \cap C_H \neq \emptyset$, an honest client $C_i \in C^* \cap C_H$ publicly raises a dispute. This is possible since all the shares of σ_i are publicly logged on $\mathcal{B}$. If the dispute is successful, all honest clients will abort the protocol. Note that a malicious client with malformed updates cannot force the protocol to abort in this way since it will not be able to produce a successful transcript with high probability (Lemma 3). If no clients raise a successful dispute, Eq. 6 follows directly from Fact 2. $\bar{C}$ represents a set of malicious clients with well-formed updates which corresponds to $C_{\mathsf{Valid}} \setminus C_H$ in Eq. 3. □

Privacy. The privacy constraint of SAVI states that nothing should be revealed about a private update u_i for an honest client C_i, except:

- u_i passes the integrity check, *i.e.*, $\mathsf{Valid}(u_i) = 1$
- anything that can be learned from the aggregate of honest clients, $\mathcal{U}_H$.

We prove that EIFFeL satisfies this privacy constraint with the help of the following two helper lemmas.

Lemma 5. *In Rounds 1-3, for an honest client* $C_i \in C_H$, *EIFFeL reveals nothing about* u_i *except* $\mathit{Valid}(u_i) = 1$.

The proof uses the fact that only m shares of C_i, which correspond to the m malicious clients, can be revealed (see full paper [22]).

Lemma 6. *In Round 4, for an honest client* $C_i \in C_H$, *EIFFeL reveals nothing about* u_i *except whatever can be learned from the aggregate.*

Proof. In Round 4, from Lemma 4 and Fact 2, the information revealed is either the aggregate or $\perp$. □

- **Setup Phase.**
  - All parties are given the security parameter κ, the number of clients n out of which at most $m < \lfloor \frac{n-1}{3} \rfloor$ are malicious, honestly generated $pp \overset{\$}{\leftarrow} \mathsf{KA.gen}(\kappa)$ and a field $\mathbb{F}$ to be used for secret sharing. Server initializes lists $\mathsf{Flag}[i] = \emptyset, i \in [n]$ and $C^* = \emptyset$.
- **Round 1 (Announcing Public Information).**

  Client: Each client C_i
  - Generates its key pair and announces the public key. $(pk_i, sk_i) \overset{\$}{\leftarrow} \mathsf{KA.gen}(pp)$, $C_i \xrightarrow{pk_i} \mathcal{B}$.

  Server:
  - Publishes the validation predicate $\mathsf{Valid}(\cdot)$. $\mathcal{S} \xrightarrow{\mathsf{Valid}(\cdot)} \mathcal{B}$
- **Round 2 (Generate and Distribute Proof).**

  Client: Each client C_i
  - Computes $n-1$ pairwise keys. $\forall C_j \in C_{\setminus i}, sk_{ij} \leftarrow \mathsf{KA.agree}(pk_j, sk_i)$
  - Generates proof $\pi_i = \big(h_i, (a_i, b_i, c_i)\big), h_i \in \mathbb{F}[X]$, $(a_i, b_i, c_i) \in \mathbb{F}^3$, $a_i \cdot b_i = c_i$ for the statement $\mathsf{Valid}(u_i) = 1$.
  - Generates shares of the input $u_i \in \mathbb{F}^d$. $\{(1, u_{i1}), \cdots, (n, u_{in}), \Psi_{u_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(u_i, [n], m+1)$
  - Generates shares of the proof π_i.
$$\{(1, h_{i1}), \cdots, (n, h_{in}), \Psi_{h_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(h_i, [n] \setminus i, m+1), \{(1, a_{i1}), \cdots, (n, a_{in}), \Psi_{a_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(a_i, [n] \setminus i, m+1)$$
$$\{(1, b_{i1}), \cdots, (n, b_{in}), \Psi_{b_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(b_i, [n] \setminus i, m+1), \{(1, c_{i1}), \cdots, (n, c_{in}), \Psi_{c_i}\} \overset{\$}{\leftarrow} \mathsf{SS.share}(c_i, [n] \setminus i, m+1)$$
  - Encrypts proof strings for all other clients. $\forall C_j \in C_{\setminus i}, \overline{(j, u_{ij})||(j, \pi_{ij})} \overset{\$}{\leftarrow} \mathsf{AE.enc}\big(sk_{ij}, (j, u_{ij})||(j, \pi_{ij})\big), \pi_{ij} = h_{ij}||a_{ij}||b_{ij}||c_{ij}$.
  - Publishes check strings and the encrypted proof strings on the bulletin. $\forall C_j \in C_{\setminus i}, C_i \xrightarrow{\overline{(j,u_{ij})||(j,\pi_{ij})}} \mathcal{B}; C_i \xrightarrow{\Psi_{u_i}, \Psi_{\pi_i}} \mathcal{B}$
- **Round 3 (Verify Proof)**.

  (i) *Verifying validity of secret shares*:

  Client: Each client C_i
  - Downloads and decrypts proof strings for all other clients from the public bulletin. Flags a client in case their decryption fails.
$$\forall C_j \in C_{\setminus i}, C_i \xleftarrow{\overline{(i,u_{ji})||(i,\pi_{ji})}, \Psi_{u_j}, \Psi_{\pi_j}} \mathcal{B}, (i, u_{ji})||(i, \pi_{ji}) \leftarrow \mathsf{AE.dec}\big(sk_{ij}, \overline{(i, u_{ji})||(i, \pi_{ji})}\big)$$
$$\perp \leftarrow \mathsf{AE.dec}\big(sk_{ij}, \overline{(i, u_{ji})||(i, \pi_{ji})}\big) \implies Cl_i \xrightarrow{\text{Flag } C_j} \mathcal{B}$$
  - Verifies the shares $u_{ji}(\pi_{ji})$ using checkstrings $\Psi_{u_j}(\Psi_{\pi_j})$ and flags all clients with invalid shares.

  Server: $\forall C_j \in C_{\setminus i}, 0 \leftarrow \big(\mathsf{SS.verify}((i, u_{ji}), \Psi_{u_j}) \wedge \mathsf{SS.verify}((i, \pi_{ji}), \Psi_{\pi_j})\big) \implies C_i \xrightarrow{\text{Flag } C_j} \mathcal{B}$
  - If client C_i flags client C_j, the server updates $\mathsf{Flag}[j] = \mathsf{Flag}[j] \cup C_i$
  - Updates the list of malicious client C^* as follows:
    - ▸ Adds all clients who have flagged $\geq m+1$ other clients. $\forall C_i$ s. t. $Z = \{j | C_i \in \mathsf{Flag}[j]\}, |Z| \geq m+1 \implies C^* \leftarrow C^* \cup C_i$
    - ▸ Adds all clients with more than $m+1$ flag reports. $|\mathsf{Flag}[i]| \geq m+1 \implies C^* \leftarrow C^* \cup C_i$
    - ▸ For clients with less flag reports, the server obtains the corresponding shares in the clear, verifies them and updates C^* accordingly. $\forall C_j$ s.t $1 \leq |\mathsf{Flag}[j]| \leq m, \forall C_i$ s.t. C_i has flagged C_j
      - $C_j \xrightarrow{(i,u_{ji}),(i,\pi_{ji})} \mathcal{B}$
      - if $\big(\mathsf{SS.verify}((i, u_{ji}), \Psi_{u_j}) \wedge \mathsf{SS.verify}((i, \pi_{ji}), \Psi_{\pi_j})\big) = 0 \implies C^* \leftarrow C^* \cup C_j$, otherwise, C_i uses the verified shares to compute its proof summary σ_{ji}
  - Publishes C^* on the bulletin. $\mathcal{S} \xrightarrow{C^*} \mathcal{B}$

  (ii) *Generation of proof summaries by the clients.*

  Server:
  - Server announces a random number $r \in \mathbb{F}$. $\mathcal{S} \xrightarrow{r} \mathcal{B}$

  Client: Each client $C_i \in C \setminus C^*$
  - Generates a summary σ_{ji} of the proof string π_{ji} based on r, $\forall C_j \in C \setminus (C^* \cup C_i), C_i \xleftarrow{r} \mathcal{B}, \sigma_{ji} = \big((i, w_{ji}^{out}), (i, \lambda_{ji})\big), C_i \xrightarrow{\sigma_{ji}} \mathcal{B}$

  (iii) *Verification of proof summaries by the server.*

  Server:
  - Downloads and verifies the proof for all clients not on C^* via robust reconstruction of the digests and updates C^* accordingly. $\forall C_i \in C \setminus C^*, \mathcal{S} \xleftarrow{\sigma_{ij}} \mathcal{B}, \big(\mathsf{SS.robustRecon}(\{(j, w_{ij}^{out})\}_{C_j \in C \setminus (C^* \cup C_i)}) \neq 1 \vee \mathsf{SS.robustRecon}(\{(j, \lambda_{ij})\}_{C_j \in C \setminus (C^* \cup C_i)}) \neq 0\big) \implies C^* \leftarrow C^* \cup C_i$
  - Publishes the updated list C^* on the bulletin. $\mathcal{S} \xrightarrow{C^*} \mathcal{B}$
- **Round 4 (Compute Aggregate).**

  Client: Each client C_i
  - If C_i is on C^*, C_i raises a dispute by sending the transcript of the reconstruction of σ_i that shows $\lambda_i = 0 \wedge w_j^{out} = 1$ and aborts, OR

    $\forall C_j \in C_{\setminus i}, C_i \xleftarrow{\sigma_{ij}} \mathcal{B}, C_i \xrightarrow{\text{Transcript of } \mathsf{SS.robustRecon}(\{(j,\sigma_{ij})\}_{C_j \in C \setminus (C^* \cup C_i)})} \mathcal{B}$
  - Aborts protocol if it sees any other client on C^* successfully raise a dispute, OR
  - If no client has raised a dispute and C_i is not on C^*, sends the aggregate of the shares of clients in $C \setminus C^*$ to the server. $\mathcal{U}_i = \sum_{C_j \in C \setminus C^*} u_{ji}$, $C_i \xrightarrow{\mathcal{U}_i} \mathcal{S}$

  Server:
  - Reconstructs the final aggregate. $\mathcal{U} \leftarrow \mathsf{SS.robustRecon}(\{(i, \mathcal{U}_i)\}_{C_i \in C \setminus C^*})$

Figure 4: EIFFeL: Description of the secure aggregation with verified inputs protocol.

6 EIFFeL Optimizations

6.1 Probabilistic Reconstruction

The Gao's decoding algorithm alongside the use of verifiable secret sharing guarantees that the correct secret will be recovered (with probability one). However, we can improve performance at the cost of a small probability of failure.

Verifying Secret Shares. As discussed in Sec. **??**, verifying the validity of the secret shares is the dominating cost for client-side computation. To reduce this cost, we propose an optimization where the validation of the shares corresponding to the proof $\pi_i = (h_i, (a_i, b_i, c_i))$ can be eliminated. Specifically, we propose the following changes to Round 3:

- Each client C_i skips verifying the validity of the shares (i, π_{ji}) for $C_j \in C_{\setminus i}$.
- Let $e = |C^*|$. The server $\mathcal{S}$ samples two sets of clients P_1, P_2 from $C \setminus \{C_i \cup C^*\}$ of size at least $3m - 2e + 1$ (P_1, P_2 can be overlapping) and performs Gao's decoding on both the sets to obtain polynomials p_1 and p_2. The server accepts the w_i^{out} (λ_i) only iff $p_1 = p_2$ and $p_1(0) = p_1(0) = 1 (p_1(0) = p_1(0) = 0)$. The cost of this step is $O(n^2 \log^2 n \log\log n)$ which is less than verifying the shares of π_i when $m < n \ll d$ (improves runtime by 2.3×, see Table 2).

Note that a $[n, k, n-k+1]$ Reed-Solomon error correcting code can correct up to $\lfloor \frac{n-k-l}{2} \rfloor$ errors with l erasures. Thus, with $m-e$ malicious clients, only $3m-2e+1$ shares are sufficient to correctly reconstruct the secret for honest clients. Since, the random sets P_1 and P_2 are not known, a malicious client with more than $m-e$ invalid shares can cheat only with probability at most $1/\binom{3m-2e+2}{n-e}$. We cannot extend this technique for the secret shares of the update u, because, unlike the value of the digests ($w^{out} = 1, \lambda = 0$), the final aggregate is unknown and needs to be reconstructed from the shares.
Improvement. Eliminates verification of check strings for the proof π_i which reduces time by 2.3× (Table 2).
Cost. Additional $1/\binom{3m-2e+2}{n-e}$ probability of failure where $e = |C^*|$.

Robust Reconstruction. In case $m \leq \sqrt{n} - 2$, the robust reconstruction mechanism can be optimized as follows. Let $q = m - |C^*|$ be the number of malicious clients that remain undetected. The server $\mathcal{S}$ partitions the set of clients in $C \setminus C^*$ into at least $q + 2$ disjoint partitions, $P = \{P_1, \cdots, P_{q+2}\}$ each of size $m + 1$. Let $p_j(x) = c_{j,0} + c_{j,1}x + c_{j,2}x^2 + \cdots + c_{j,m}x^m$ represent the polynomial corresponding to the $m + 1$ shares of partition P_j. Recall that recovering just $p_j(0) = c_{j,0}$ suffices for a typical Shamir secret share reconstruction. However, now, the server $\mathcal{S}$ recovers the entire polynomial p_j, *i.e.*, all of its coefficients $\{c_{j,0}, c_{j,1}, \cdots, c_{j,q}\}$ for all $q + 2$ partitions. Based on the pigeon hole principle, it can be argued that at least two of the partitions ($P_l, P_k \in P$) will consist of *honest* clients only. Hence, we must have at least two polynomials p_l and p_k that match and the value of the secret is their constant coefficient $p_l(0)$. Note that the above mentioned optimization of skipping verifying the shares of the proof can be applied here as well. A malicious client can cheat (*i.e.*, make the server $\mathcal{S}$ accept even when $w_i^{out} \neq 1 \vee \lambda_i \neq 0$ or reject the proof for an honest client) only if they can manipulate the shares of at least two partitions which must contain at least $2(m + 1) - q$ honest clients. Since the random partition P is not known to the clients, this can happen only with probability $1/\binom{2(m+1)-q}{n-m-1}$.
Improvement. Reduces the number of polynomial interpolations.
Cost. Additional $1/\binom{2(m+1)-q}{n-m-1}$ probability of failure where $q = m - |C^*|$.

6.2 Crypto-Engineering Optimizations

Equality Checks. The equality operator = is relatively complicated to implement in an arithmetic circuit. To circumvent this issue, we replace any validation check of the form $\Phi(u) = c_1 \vee \Phi(u) = c_2 \vee \cdots \vee \Phi(u) = c_k$ in the output nodes of $\mathsf{Valid}(\cdot)$, where $\Phi(\cdot)$ is some arithmetic function, by an output of the form $(\Phi(u) - c_1) \times \cdots \times (\Phi(u) - c_k)$. Recall that in EIFFeL, the honest clients have well-formed inputs that satisfy $\mathsf{Valid}(\cdot)$ by definition. Hence, this optimization does not violate the privacy of honest, which is our security goal.
Improvement. Reduces the circuit size $|\mathsf{Valid}|$.
Cost. No cost.

Proof Summary Computation. In addition to being a linear secret sharing scheme, Shamir's scheme is also multiplicative: given the shares of two secrets (i, z_i) and (i, v_i), a party can locally compute (i, s_i) with $s = z \cdot v$. However, if the original shares correspond to a polynomial of degree t, the new shares represent a polynomial of degree $2t$. Hence, we do not rely on this property for the multiplication gates of $\mathsf{Valid}(\cdot)$ as it would support only limited number of multiplications. However, if $m < \frac{n-1}{4}$, we can still leverage the multiplicative property to generate shares of the random digest $\lambda_i = f_i(r) \cdot g_i(r) = h_i(r)$ locally (instead of using Beaver's triples).
Improvement. Saves a round of communication and reduces the number of robust reconstructions for λ_i from three to just one (details in the full paper [22]).
Cost. No cost.

Random Projection. As shown in Table 1, both communication and computation grows linearly with the data dimension d. Hence, we rely on the random projection [54] technique for reducing the dimension of the updates. Specifically, we use the fast random projection using Walsh-Hadamard transforms [4].
Improvement. Reduces the data dimension which helps both computation and communication cost.
Cost. Empirical evaluation (Sec. 7.2) shows that the efficacy of $\mathsf{Valid}(\cdot)$ is still preserved.

7 Experimental Evaluation

7.1 Performance Evaluation.

In this section, we analyze the performance of EIFFeL.

Configuration. We run experiments on two Amazon EC2 c5.9large instances with Intel Xeon Platinum 8000 processors. To emulate server-client communication, we use two instances in the US East (Ohio) and US West (Oregon) regions, with a round trip time of 21 ms. We implemented EIFFeL in Python and C++ using NTL library [1]. We use AES-GCM for encryption, a 56-bit prime field $\mathbb{F}$ and probabilistic quantization [38]. For key agreement, we use elliptic curve Diffie-Hellman [28] over the NIST P-256 curve. Unless otherwise specified, the default settings are $d = 1K$, $n = 100$, $m = 10\%$ and $|\mathsf{Valid}(\cdot)| \approx 4d$. We report the mean of 10 runs for each experiment.

Computation Costs. Fig. 5 presents EIFFeL's runtime. We vary the number of malicious clients between 5%-20% of the number of

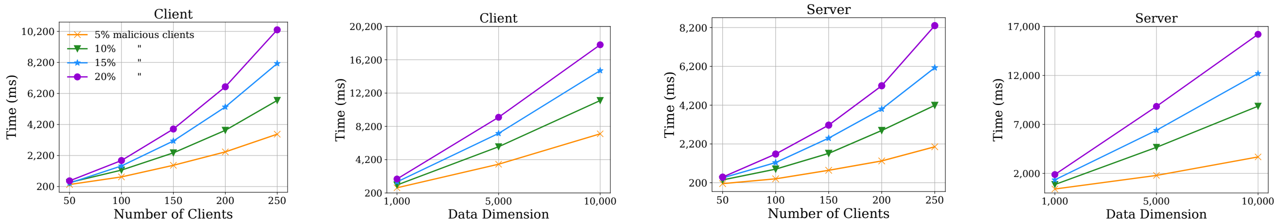


Figure 5: Computation cost analysis of EIFFeL. The left two plots show the runtime of a single client client in milliseconds as a function of: (left) the number of clients n and (right) dimensionality of the updates d. The right two plots show the runtime of the server as a function of the same variables. The results demonstrate that performance decays quadratically in n, and linearly in d.

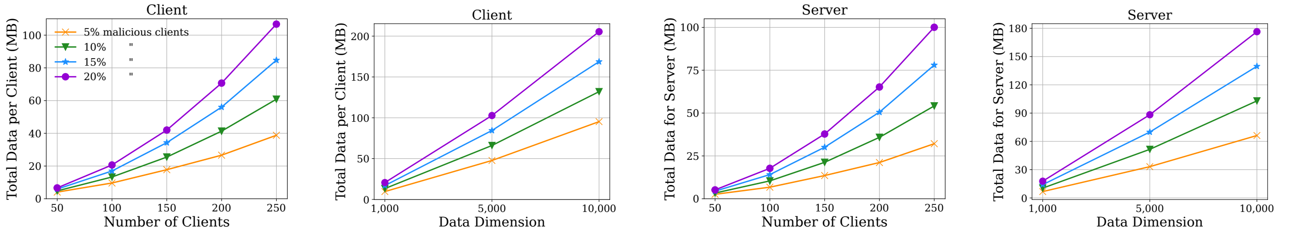


Figure 6: Communication cost analysis of EIFFeL. The left two plots show the amount of communication (in MB) for each client as a function of: (left) the number of clients n and (right) dimensionality of the updates d. The right two plots show the the amount of communication (in MB) for the server as a function of the same variables. The results show communication increases quadratically in n, and linearly in d.

clients. We observe that per-client runtime of EIFFeL is low: it is 1.3s if $m = 10\%$, $d = 1K$, and $n = 100$. The runtime scales quadratically in n because a client has $O(mnd)$ computation complexity (see Table 1) and m is a linear function of n. As expected, the runtime increases linearly with d. A client takes around 11s when $d = 10K$, $n = 100$, and $m = 10\%$. The runtime for the server is also low: the server completes its computation in about 1s for $n = 100$, $d = 1K$, and $m = 10\%$. The server's runtime also scales quadratically in n due to the $O(mnd)$ computation complexity (Table 1). The runtime increases linearly with d.

In Fig. 7, we break down the runtime per round. We observe that: Round 1 (announcing public information) incurs negligible cost for both clients and the server; and Round 3 (verify proof) is the costliest round for both clients and the server where the dominating cost is verifying the validity of the shares (Sec. **??**). Note that the server has no runtime cost for Round 2 since the proof generation only involves clients.

# Clients (n)	Time (ms)	Improvement over Unoptimized EIFFeL	 BREA [65]
50	1,072	2.3×	2.5×
100	2,367	2.3×	5.2×
150	4,326	2.3×	7.8×
200	6,996	2.3×	12.8×
250	10,389	2.3×	18.5×

Table 2: End-to-end time for a single iteration of EIFFeL with $d = 1000$ and $m = 10\%$ malicious clients, as a function of the number of clients, n. We also compare it with a variant of EIFFeL without optimizations, and with BREA [65].

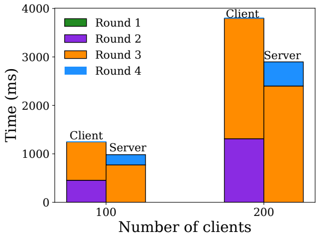
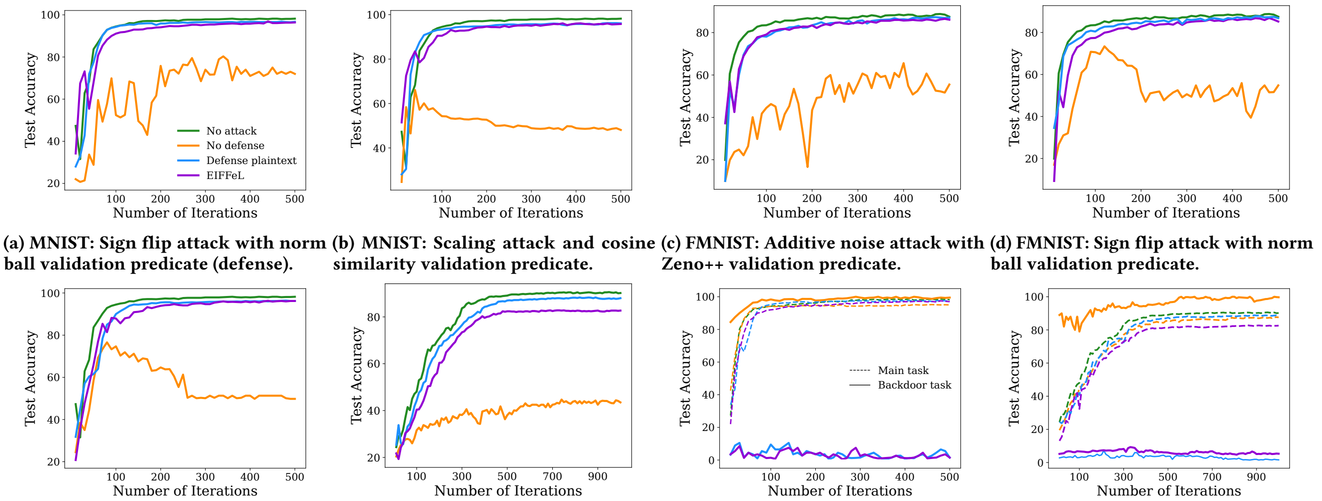


Figure 7: Computation cost per round in EIFFeL.

Table 2 presents our end-to-end performance which contains the runtimes of a client, the server and the communication latencies. For instance, the end-to-end runtime for $n = 100$, $d = 1K$ and $m = 10\%$ is $\sim 2.4s$. We also present the impact of one of our key optimizations – eliminating the verification of the secrets shares of the proof – which cuts down the costliest step in EIFFeL and improves the performance by 2.3×. Additionally, we compare EIFFeL's performance with BREA [65], which is a Byzantine-robust secure aggregator. EIFFeL differs from BREA in two key ways: (1) EIFFeL is a general framework for per-client update integrity checks whereas BREA implements the multi-Krum aggregation algorithm [14] that considers the entire dataset to determine the malicious updates (computes all the pairwise distances between the clients and then, detects the outliers), and (2) BREA has an additional privacy leakage as it reveals the values of all the pairwise distances between clients. Nevertheless, we choose BREA as our baseline because, to the best of our knowledge, this is the only prior work that: (1) detects and removes malformed updates, and (2) works in the malicious threat model with (3) a single server (see Table 3, Sec. 9). We observe that EIFFeL outperforms BREA and that the improvement increases with n. For instance, for $n = 250$, EIFFeL is 18.5× more performant than BREA. This is due to BREA's complexity of $O(n^3 \log^2 n \log\log n + mnd)$, where the $O(n^3)$ factor is due to each client partaking in the computation of the $O(n^2)$ pairwise distances.

Communication Cost. Fig. 6 depicts the total data transferred by a client and the server. The communication complexity is $O(mnd)$ for a single client and for the server. Hence, the total communication increases quadratically with n and linearly with d, respectively. We observe that EIFFeL has acceptable communication cost. For instance, the total data consumed by a client is 132MB for the configuration $n = 100, d = 10K, m = 10\%$. This is equivalent to streaming a full-HD video for 26s [2]. Since most clients partake in FL training



(a) MNIST: Sign flip attack with norm ball validation predicate (defense). **(b) MNIST: Scaling attack and cosine similarity validation predicate.** **(c) FMNIST: Additive noise attack with Zeno++ validation predicate.** **(d) FMNIST: Sign flip attack with norm ball validation predicate.**

(e) MNIST: Min-Max attack with Zeno++ validation predicate. **(f) CIFAR-10: Min-Sum attack with cosine similarity validation predicate.** **(g) EMNIST: Backdoor Attack-1 with norm bound validation predicate.** **(h) CIFAR-10: Backdoor Attack-2 with norm bound validation predicate.**

Figure 8: Accuracy analysis of EIFFeL. Test accuracy is shown as a function of the FL iteration for different datasets and attacks.

iterations infrequently, this communication is acceptable.

Note. Recall, we assume the size of the validation predicate to be $|\mathsf{Valid}| = O(d)$ since $\mathsf{Valid}(\cdot)$ defines a function on the input which is d-dimensional. This assumption is validated by the state-of-the-art predicates tested in Sec. 7.2. The above experiments use $|\mathsf{Valid}| \approx 4d$. Hence, the overall complexity (see full paper [22]) is dominated by the $O(mnd)$ term and does not depend on the validation predicate.

7.2 Integrity Guarantee Evaluation

In this section, we evaluate EIFFeL's efficacy in ensuring update integrity on real-world datasets.

Datasets. We evaluate EIFFeL on three image datasets:

- *MNIST* [42] is a digit classification dataset of $60K$ training images and $10K$ test images with ten classes.
- *EMNIST* [23] is a writer-annotated handwritten digit classification dataset with $\sim 340K$ training and $\sim 40K$ testing images.
- *FMNIST* [77] is identical to MNIST in terms number of classes, and number of training and test images.
- *CIFAR-10* [40] contains RGB images with ten object classes. It has $50K$ training and $10K$ test images.

Models. We test EIFFeL with three classification models:

- *LeNet-5* [41] has five layers and $60K$ parameters, and is used to experiment on MNIST and EMNIST.
- For FMNIST, we use a five-layer convolutional network with $70K$ parameters and a similar architecture as LeNet-5.
- We use *ResNet-20* [34] with 20 layers and $273K$ parameters for CIFAR-10.

Validation Predicates. To demonstrate the flexibility of EIFFeL, we evaluate four validations predicates, which represent the current *state-of-the-art* defenses against data poisoning, as follows:

- *Norm Bound* [69]. This method checks whether the ℓ_2-norm of a client update is bounded: $\mathsf{Valid}(u) = \mathbb{I}[\|u\|_2 < \rho]$ where $\mathbb{I}[\cdot]$ is the indicator function and the threshold ρ is computed from the public dataset $\mathcal{D}_P$.
- *Norm Ball* [67]. This method checks whether a client update is within a spherical radius from v which is the gradient update computed from the clean public dataset $\mathcal{D}_P$: $\mathsf{Valid}(u) = \mathbb{I}\big[\|u - v\|_2 \leq \rho\big]$ where radius ρ is also computed from $\mathcal{D}_P$.
- *Zeno++* [72] compares the client update with a loss gradient v that is computed on the public dataset $\mathcal{D}_P$: $\mathsf{Valid}(u) = \mathbb{I}[\gamma\langle v, u\rangle - \rho||u||_2 \geq -\gamma\epsilon]$ where γ, ρ and ϵ are threshold parameters also computed from $\mathcal{D}_P$ and u is ℓ_2-normalized to have the same norm as v.
- *Cosine Similarity* [5, 19]. This method compares the cosine similarity between the client update u and the global model update of the last iteration u': $\mathsf{Valid}(u) = \mathbb{I}\big[\frac{\langle u,u'\rangle}{\|u\|_2\|u'\|_2} < \rho\big]$ where ρ is computed from $\mathcal{D}_P$ and u is ℓ_2-normalized to match norm of u'.

Poisoning Attacks. To test the efficacy of EIFFeL's implementations of the four validation predicates introduced above, we test it against seven poisoning attacks:

- *Sign Flip Attack* [27]. In this attack, the malicious clients flip the sign of their local update: $\hat{u} = -c \cdot u, c \in \mathbb{R}_+$.
- *Scaling Attack* [10] scales a local update to increase its influence on the global update: $\hat{u} = c \cdot u, c \in \mathbb{R}_+$.
- *Additive Noise Attack* [43] adds Gaussian noise to the local update: $\hat{u} = u + \eta, \eta \sim \mathcal{N}(\sigma, \mu)$.
- *Min-Max Attack* [62] sets all the malicious updates to be: $\text{argmax}_\gamma \max_{i\in[n]} ||\hat{u} - u_i||_2 \leq \max_{i,j\in[n]} ||u_i - u_j||_2; \hat{u} = \frac{1}{n}\sum_{i=1}^{n} u_i + \gamma \cdot u^p$, where u^p is a dataset optimized perturbation vector. Here, the adversary is assumed to have access to the benign (well-formed) updates of *all* clients. This attack finds the malicious gradient whose maximum distance from a benign gradient is less than the maximum distance between any two benign gradient.
- *Min-Sum Attack* [62] sets all the malicious updates to be: $\text{argmax}_\gamma \sum_{i\in[n]} ||\hat{u} - u_i||_2 \leq \max_{i\in[n]} \sum_{j\in[n]} ||u_i - u_j||_2; \hat{u} = \frac{1}{n}\sum_{i=1}^{n} u_i + \gamma \cdot u^p$, where u^p is a dataset optimized perturbation vector. Here, the adversary is assumed to have access to the benign updates of *all* clients. This attack finds the malicious gradient such that the sum

of its distances from all the other gradients is less than the sum of distances of any benign gradient from other benign gradients.
- *Backdoor Attack-1* [69] classifies the digit seven as the digit one for EMNIST.
- *Backdoor Attack-2* [5] classifies images of green cars as birds for CIFAR-10.

Configuration. We use the same configuration as before. We implement the image-classification models in PyTorch. We randomly select 10K samples from each training set as the public dataset $\mathcal{D}_P$ and train on the remaining samples. EMNIST is collected from 3383 clients with $\sim$ 100 images per client. For all other datasets, the training set is divided into 5K subsets to create the local dataset for each client. For each training iteration, we sample the required number of data subsets out of these 5K subsets.

Results. Fig. 8 shows the accuracy of different image-classification models in EIFFeL. We set $n = 100$ and $m = 10\%$, and use random projection to project the updates to a dimension d of 1K (MNIST, EMNIST), 5K (FMNIST), or 10K (CIFAR-10). For the two backdoor attacks, we consider $m = 5\%$. Our experiment assesses how the random projection affects the efficacy of the integrity checks. We observe that for MNIST (Figs. 8a, 8b and 8e), EMNIST (Fig. 8g) and FMNIST (Fig. 8c and 8d), EIFFeL achieves performance comparable to a baseline that applies the defense (validation predicate) on the plaintext. In most cases, the defenses retain their efficacy even after random projection. This is because they rely on computing inner products and norms of the update; these operations preserve their relative values after the projection with high probability [54]. We observe a drop in accuracy ($\sim$ 7%) on CIFAR-10 (Figs. 8f and 8h) as updates for ResNet-20 with 273K parameters are projected to 10K. The end-to-end per-iteration time ($m = 10\%$) for MNIST, EMNIST, FMNIST, and CIFAR-10 is $2.4s$ (Table 2), $2.4s$, $10.7s$, and $20.5s$, respectively. The associated communication costs for the client are 13.3MB, 13.3MB, 65.8MB, and 132MB (Fig. 6). Additional evaluation results are presented in the full paper [22].

8 Discussion

In this section, we discuss possible avenues for future research (additional discussion is in the full paper [22]).

Handling Higher Fraction of Malicious Clients. For $\lfloor \frac{n-1}{3} \rfloor < m < \lfloor \frac{n-1}{2} \rfloor$ (honest majority), the current implementation of EIFFeL can detect but not remove malformed inputs (Gao's decoding algorithm returns $\perp$ if $m > \lfloor \frac{n-1}{3} \rfloor$). Robust reconstruction in this case could be done via Guruswami-Sudan list decoder [48]. We do not do so in EIFFeL because the reconstruction might fail sometimes.

Handling Client Dropouts. In practice, clients might have only sporadic access to connectivity and so, the protocol must be robust to clients dropping out. EIFFeL can already accommodate malicious client dropping out – it is straightforward to extend this for the case of honest clients as well.

Towards poly-logarithmic complexity. Currently, dominant term in the complexity is $O(mnd)$ which results in a $O(n^2)$ dependence on n (since we consider m is a fraction of n). This can be reduced to $O(n \log^2 nd)$ by using the techniques from [9]. A detailed discussion is presented in App. **??**.

9 Related Work

Table 3: Comparison of EIFFeL with Related Work

Work	Malicious Threat Model	Single Server	Removes Malformed Inputs	Arbitrary Integrity Checks
He et.al [35]	×	×	×	×
FLGuard [55]	×	×	×	×
RoFL [18]	×	✓	×	×
BREA* [65]	✓	✓	✓	×
EIFFeL(Our)	✓	✓	✓	✓

*Has additional privacy leakage

Secure Aggregation. Prior work has addressed the problem of (non-Byzantine) secure aggregation in FL [3, 9, 15, 66]. A popular approach is to use pairwise random masking to protect the local updates [3, 15]. Advancements have been made in the communication overhead [16, 39, 66].

Robust Machine Learning. A large number of studies have explored methods to make machine learners robust to Byzantine failures [5, 10, 36]. Many of these robust machine-learning methods require the learned to have full access to the training data or to fully control the training process [25, 33, 46, 64, 67, 70] which is infeasible in FL. Another line of work has focused on the development of estimators that are inherently robust to Byzantine errors [14, 20, 56, 57, 75]. In our work, we target a set of methods that provides robustness by checking per-client updates [14, 31, 63].

Verifying Data Integrity in Secure Aggregation. Table 3 compares EIFFeL with prior work. There are three key differences between RoFL [18] and EIFFeL: (1) RoFL is designed only for range checks with ℓ_2 or ℓ_∞ norms. Specifically, RoFL uses Bulletproofs which is especially performant for range proofs (range proofs can be aggregated where one can prove that n commitments lie within a given range by providing only an additive $O(log(n))$ group elements over the length of a single proof). RoFL's performance is primarily based on this aspect of Bulletproof and all of its optimizations work only for range proofs. As such RoFL cannot support any other checks with the same performance as currently reported in the paper. By contrast, EIFFeL is a general framework that supports arbitrary validation predicates with good performance. (2) RoFL is susceptible to DoS attacks because it *only* detects malformed updates and aborts if it finds one. Specifically, the recovery of the final aggregate in RoFL requires a step of nonce cancellation that involves all the inputs by design. Hence, even if one of the input is invalid, the final aggregate will be ill-formed. By contrast, EIFFeL is a SAVI protocol that detects and removes malformed updates in every round. (3) RoFL assumes an honest-but-curious server, whereas EIFFeL considers a malicious threat model. BREA [65] also removes outlying updates but, unlike EIFFeL, it leaks pairwise distances between inputs. Alternative solutions [35, 55] for distance-based Byzantine-robust aggregation uses two non-colluding servers in the semi-honest threat model, which is incompatible with FL.

10 Conclusion

Practical FL settings need to ensure both the privacy and integrity of model updates provided by the clients. In this paper, we have formalized these goals in a new protocol, SAVI, that securely aggregates *only* well-formed inputs (*i.e.*, updates). To demonstrate the feasibility of SAVI, we have proposed EIFFeL: a system that efficiently instantiates a SAVI protocol.

References

[1] https://libntl.org/.

[2] Youtube system requirements. https://support.google.com/youtube/answer/78358?hl=en.

[3] Gergely Ács and Claude Castelluccia. I have a dream! differentially private smart metering. In *Proceedings of the 13th International Conference on Information Hiding*, IH'11, page 118–132, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 557–563, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *arXiv:1807.00459*, 2018.

[6] Raef Bassily, Albert Cheu, Shay Moran, Aleksandar Nikolov, Jonathan Ullman, and Zhiwei Steven Wu. Private query release assisted by public data. In *ICML*, 2020.

[7] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[8] Amos Beimel, Aleksandra Korolova, Kobbi Nissim, Or Sheffet, and Uri Stemmer. The power of synergy in differential privacy: Combining a small curator with local randomizers. In *ITC*, 2020.

[9] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1253–1269, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *Proceedings of the International Conference on Machine Learning*, pages 634–643, 2019.

[11] Abhishek Bhowmick, John C. Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan M. Rogers. Protection against reconstruction and its applications in private federated learning. *ArXiv*, abs/1812.00984, 2018.

[12] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the International Coference on International Conference on Machine Learning*, pages 1467–1474, 2012.

[13] Richard E. Blahut. Theory and practice of error control codes. 1983.

[14] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems*, pages 118–128, 2017.

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

[16] Keith Bonawitz, Fariborz Salehi, Jakub Konecný, H. Brendan McMahan, and Marco Gruteser. Federated learning with autotuned communication-efficient secure aggregation. *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pages 1222–1226, 2019.

[17] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, oct 1985.

[18] Lukas Burkhalter, Hidde Lycklama, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. Rofl: Attestable robustness for secure federated learning. In *arXiv:2107.03311*, 2021.

[19] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. 2021.

[20] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Draco: Byzantine-resilient distributed training via redundant gradients. In *Proceedings of the International Conference on Machine Learning*, 2018.

[21] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. In *arXiv:1712.05526*, 2017.

[22] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Full paper, 2021.

[23] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926, 2017.

[24] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[25] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *IEEE Symposium on Security and Privacy (SP)*, pages 81–95, 2008.

[26] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 317–334, USA, 2009. USENIX Association.

[27] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, and Mahsa Taziki. Asynchronous byzantine machine learning (the case of sgd). In *ICML*, 2018.

[28] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[29] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[30] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438, 1987.

[31] Clement Fung, Chris J.M. Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning. In *arXiv:1808.04866*, 2018.

[32] Shuhong Gao. *A New Algorithm for Decoding Reed-Solomon Codes*, pages 55–68. Springer US, Boston, MA, 2003.

[33] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *arXiv:1708.06733*, 2017.

[34] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[35] Lie He, Sai Praneeth Karimireddy, and Martin Jaggi. Secure byzantine-robust machine learning, 2020.

[36] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurelien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G.L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adria Gascon, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konecny, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrede Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Ozgur, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramer, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. In *arXiv:1912.04977*, 2019.

[37] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[38] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *CoRR*, abs/1610.05492, 2016.

[39] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *ArXiv*, abs/1610.05492, 2016.

[40] Alex Krizhevsky. The cifar-10 dataset.

[41] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[42] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits.

[43] Liping Li, Wei Xu, Tianyi Chen, Georgios Giannakis, and Qing Ling. Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:1544–1551, 07 2019.

[44] Suyi Li, Yong Cheng, Wei Wang, Yang Liu, and Tianjian Chen. Learning to detect malicious clients for robust federated learning. *CoRR*, abs/2002.00211, 2020.

[45] Shu Lin and Daniel J. Costello. *Error control coding: fundamentals and applications*. Pearson/Prentice Hall, Upper Saddle River, NJ, 2004.

[46] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. pages 273–294, 2018.

[47] Terrance Liu, Giuseppe Vietri, Thomas Steinke, Jonathan Ullman, and Zhiwei Steven Wu. Leveraging public data for practical private query release, 2021.

[48] R. J. McEliece. The guruswami–sudan decoding algorithm for reed–solomon codes, 2003.

[49] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data, 2017.

[50] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2017.

[51] Shike Mei and Xiaojin Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2871–2877, 2015.

[52] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 691–706, 2019.

[53] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019.

[54] Jelani Nelson. Sketching algorithms.

[55] Thien Duc Nguyen, Phillip Rieger, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Flguard: Secure and private federated learning, 2021.

[56] Xudong Pan, Mi Zhang, Duocai Wu, Qifan Xiao, Shouling Ji, and Zhemin Yang. Justinian's GAAvernor: Robust distributed learning with gradient aggregation agent. In *USENIX Security*, pages 1641–1658, 2020.

[57] Shashank Rajput, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Detox: A redundancy-based framework for faster and more robust gradient aggregation. 2019.

[58] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 196–210, New York, NY, USA, 2019. Association for Computing Machinery.

[59] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[60] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[61] Virat Shejwalkar and Amir Houmansadr. Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning. In *NDSS*, 2021.

[62] Virat Shejwalkar and Amir Houmansadr. Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning. In *NDSS*, 2021.

[63] Shiqi Shen, Shruti Tople, and Prateek Saxena. Auror: Defending against poisoning attacks in collaborative deep learning systems. In *ACM ACSAC*, pages 508–519, 2016.

[64] Yanyao Shen and Sujay Sanghavi. Learning with bad training data via iterative trimmed loss minimization. In *International Conference on Machine Learning (ICML)*, pages 5739–5748, 2019.

[65] Jinhyun So, Basak Guler, and A. Salman Avestimehr. Byzantine-resilient secure federated learning. *IEEE Journal in Selected Areas in Communications: Machine Learning in Communications and Networks*, 2020.

[66] Jinhyun So, Basak Guler, and A. Salman Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning, 2021.

[67] Jacob Steinhardt, Pang Wei W. Koh, and Percy S. Liang. Certified defenses for data poisoning attacks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3517–3529, 2017.

[68] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H. Brendan McMahan. Can you really backdoor federated learning? In *arXiv:1911.07963*, 2019.

[69] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H. Brendan McMahan. Can you really backdoor federated learning? *ArXiv*, abs/1911.07963, 2019.

[70] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 707–723, 2019.

[71] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. Dba: Distributed backdoor attacks against federated learning. In *ICLR*, 2020.

[72] Cong Xie. Zeno++: robust asynchronous SGD with arbitrary number of byzantine workers. *CoRR*, abs/1903.07020, 2019.

[73] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. In *Proceedings of the International Conference on Machine Learning*, 2019.

[74] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno++: Robust fully asynchronous SGD. In *Proceedings of the International Conference on Machine Learning*, 2020.

[75] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *International Conference on Machine Learning (ICML)*, 2019.

[76] Hongxu Yin, Arun Mallya, Arash Vahdat, José Manuel Álvarez, Jan Kautz, and Pavlo Molchanov. See through gradients: Image batch recovery via gradinversion. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 16332–16341, 2021.

[77] Zalando. Fashion mnist.

[78] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In *NeurIPS*, 2019.

[79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, EUROSAM '79, page 216–226, Berlin, Heidelberg, 1979. Springer-Verlag.