Learning from Failures: Secure and Fault-Tolerant Aggregation for Federated Learning

Mohamad Mansouri
EURECOM & Thales SIX GTS, France
mohamad.mansouri@eurecom.fr

Melek Önen
EURECOM, France
melek.onen@eurecom.fr

Wafa Ben Jaballah
Thales SIX GTS, France
wafa.benjaballah@thalesgroup.com

Abstract

Federated learning allows multiple parties to collaboratively train a global machine learning (ML) model without sharing their private datasets. To make sure that these local datasets are not leaked, existing work propose to rely on a secure aggregation scheme that allows parties to encrypt their model updates before sending it to the central aggregator. In this work, we design and evaluate a new secure and fault-tolerant aggregation scheme for federated learning that is robust against client failures. We first propose a threshold-variant of the secure aggregation scheme proposed by Joye and Libert in order to support some client failures. Using this new building block together with dedicated decentralized key management scheme and a dedicated input encoding solution, we design a privacy-preserving federated learning protocol that, when executed among n clients, can recover from up to $\frac{n}{3}$ failures. Our solution is secure against a malicious aggregator who can manipulate messages to learn clients' individual inputs. We show that our solution outperforms the state of the art fault-tolerant secure aggregation schemes in terms of computation cost on both the client and the server sides. For example, with ML models of 100,000 parameters, 600 clients and at most 180 client failures, our protocol is x4.6 faster at the client and x1.3 faster at the server.

1 Introduction

Machine learning (ML) nowadays plays a very important role in various domains such as autonomous cars, healthcare systems, recommendation systems, etc. The efficiency and accuracy of ML models usually rely on the processing of very large amount of data that are collected from multiple data sources and that are often privacy-sensitive. To cope with the privacy protection of such data, the federated learning paradigm has recently emerged: Federated learning can be defined as a collaborative ML technique whereby multiple clients obtain a joint model by locally training the model on their private dataset and sending parameter updates to the aggregator, only. Although being promising, as shown in [24, 25], a naive use of such a scheme can still result in some leakage based on the exchanged parameters and thus model updates also need to be protected.

Secure aggregation [19, 26] which ensures the aggregation of multiple parties' inputs without disclosing them individually, becomes a common solution to address this problem in federated learning. The server receiving protected model updates from clients is still able to compute their aggregate. Unfortunately, the majority of secure aggregation solutions requires all clients to be online. If a client fails to provide its protected input the server will not be able to compute the aggregated result.

In federated learning applications, clients are often mobile devices that may frequently encounter failures and hence existing secure aggregation solutions fall short to address such a problem. A previous work from Bonawitz et al. [4] develops a fault-tolerant secure aggregation that enables the server to recover the aggregate from up to t out of n client failures. The authors design their solution based on a secure masking scheme [9]: clients use one-time-pad encryption (i.e., modular addition) with a unique mask to protect their inputs. The masks are chosen such that their sum is zero and are obtained through the Diffie-Hellman (DH) key exchange scheme [8] protocol run among each pair of clients. Additionally, DH keys are shared among n clients using Shamir's secret sharing [31] in order to recover them in case of client failures. This scheme has been used as a building block for a significant number of privacy-preserving federated learning solutions [2, 3, 5, 14, 16, 18, 34–36, 40].

Unfortunately, this solution still incurs a significant computation and communication overhead originating from the execution of the DH key exchange protocol among each pair of clients and the generation of the mask at each FL round. This is mainly due to the fact that the mask can only be used once. We therefore propose a new solution that enables a client to use a same key for multiple FL rounds. We propose to revisit the Joye-Libert (JL) solution proposed in [17] in order to cope with client failures and further make use of

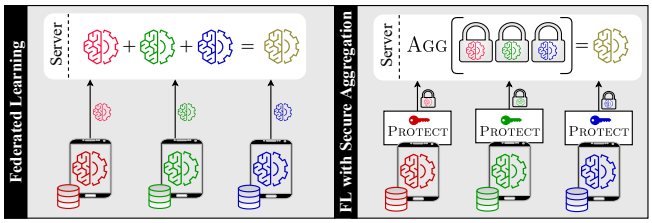


Figure 1: Illustration of federated learning (with secure aggregation in the right side of the figure . The clients (e.g., smartphones) locally train the model on their own private dataset. The clients send their trained models (protected when using secure aggregation) to the server. The server aggregates the models (the server learns only the aggregate when using secure aggregation).

it to design a fault-tolerant and secure aggregation scheme dedicated to federated learning applications. The new version of the scheme allows the sharing the client's individual keys using Shamir's secret sharing [31] so that when some clients fail to submit their protected inputs, t out of n clients use the key shares of the failed clients' to provide a ciphertext representing the protected zero-value of the failed clients. The aggregator can use the protected zero-value to correctly aggregate the inputs of the online users. Compared to [4], we provide a more efficient fault-tolerant secure aggregation scheme since it does not require to redistribute the protection keys for each FL round.

Our contributions can be summarized as follows:

- We design a new version of the Joye-Libert secure aggregation scheme [17] by allowing clients to secret share their individual keys. The new scheme is fault-tolerant and suitable for federated learning applications. It completes the aggregation in two communication rounds (among the server and the clients) which is less than in the solution [4] (four communication rounds).
- We implement a prototype of our protocol and evaluate it through a comparative study with the protocol in [4].
- Through our experimental study we show that our solution outperforms the solution in [4] in terms of computation at the client side and supports more failures with the same copmutational cost at the server side.
- We provide a theoretical security analysis of our protocol and we show that our protocol is secure in both the honest-but-curious and the active adversary settings.

2 Secure Aggregation for Federated Learning

In federated learning (FL), a set $\mathcal{U}$ of FL clients locally train a joint machine learning model $\mathcal{M}$. Each client $u \in \mathcal{U}$ uses it own private training dataset $\mathcal{D}_u$ to train the model. At Fl round τ, a FL client runs the training algorithm (eg., Stochastic Gradient Descent (SGD) [6]) on the current model $\mathcal{M}_\tau$. As a result of the training, it locally updates the model $\mathcal{M}_{u,\tau+1} \leftarrow train(\mathcal{M}_\tau, \mathcal{D}_u)$. All the FL clients send the updated models to the FL server which aggregates them by computing their average:

$$\mathcal{M}_{\tau+1} \leftarrow \frac{\sum_{u=1}^{n} \mathcal{M}_{u,\tau+1}}{n}$$

Finally, the FL server sends the aggregated model $\mathcal{M}_{\tau+1}$ to all clients, then a new FL round starts. The process is repeated until the model $\mathcal{M}$ is learned after T rounds.

Although FL clients keep their own datasets private, studies has shown that adversaries who have access to the client's updated model $\mathcal{M}_{u,\tau+1}$ can infer information about its private dataset $\mathcal{D}_u$ [24, 25]. Hence the local models should remain confidential even against the FL server. Secure aggregation solves this problem by protecting the locally trained models of the FL clients. Figure 1 demonstrates a FL protocol with and without secure aggregation. Secure aggregation involves two roles: The "user" represents the FL client, and the "aggregator" represents the FL server. We consider the following threat model.

Threat model The assumptions on the adversaries are summarized as follows:

- *For the Aggregator*: We consider two adversary settings for the aggregator: (i) an **honest-but-curious model** where the aggregator does not modify its inputs to the protocol but it still tries to learn private information about clients local models. (ii) an **active model** where the aggregator may manipulate the messages in order to learn the users' private information.
- *For the Users*: We assume that, up to $n-t$ users may collude with each other, and/or with the aggregator, where t is a security parameter representing the number of honest users. Colluding users share their private information. We do not consider malicious users that try to manipulate the aggregation outcome or users that perform denial-of-service attacks.

Secure Aggregation A secure aggregation scheme consists of three probabilistic polynomial time (PPT) algorithms:

- $(k_a, \{k_u\}_{\forall u \in \mathcal{U}}, pp) \leftarrow$ **Setup**(λ): Given a security parameter λ, this algorithm generates the public parameters pp, the users' keys $\{k_u\}_{\forall u \in \mathcal{U}}$ and the aggregator's key k_a.
- $y_{u,\tau} \leftarrow$**Protect**$(pp, k_u, \tau, x_{u,\tau})$: For time period τ, user u applies the protection algorithm on its input $x_{u,\tau}$ using its own secret key k_u. It outputs the ciphertext $y_{u,\tau}$.
- $X_\tau \leftarrow$ **Agg**$(pp, k_a, \{y_{u,\tau}\}_{\forall u \in \mathcal{U}})$: This aggregation algorithms run by aggregator outputs the aggregate results

($X_t = \sum_{u=1}^{N} x_{u,\tau}$) given the individual ciphertexts of all users and the aggregator's key k_a.

A secure aggregation scheme is considered secure in the honest-but-curious model if it ensures *Aggregator Obliviousness (AO)*.

Aggregator Obliviousness (AO) This security notion ensures that an honest-but-curious aggregator cannot learn more than the sum X_t of the users inputs. If some users are corrupted (i.e., users sharing their private information with the aggregator), the notion only requires that the aggregator gets no extra information about the values of the honest users beyond that their aggregate value. Furthermore, it is assumed that each user protects only one value per time period. In [17], the authors provide a formalization of this security notion in a challenger-attacker game. Notice that this security notion does not ensures the correctness of the aggregation result. It only ensures the privacy of the inputs.

Problem statement The basic definition of secure aggregation, assumes the availability of all users. This is because the aggregation algorithm (**Agg**) requires the ciphertexts of all users $\{y_{u,\tau}\}_{\forall u \in \mathcal{U}}$. When some FL clients drop from the protocol (due to network problems or to failures), they are not able to provide their protected inputs $y_{u,\tau}$. Hence, the aggregator is not able to collect all ciphertexts $\{y_{u,\tau}\}_{\forall u \in \mathcal{U}}$. This calls for an efficient solution that allows the aggregate to be recovered, even if some clients failed to send their protected inputs.

3 Preliminaries

In this section, we present the main cryptographic primitives that are used as building blocks for our protocol.

3.1 Pseudo Random Generator

$B \leftarrow \mathbf{PRG}(b)$: is a pseudo random generator that can extend a seed $b \in \mathbb{Z}$ to a vector $B \in \mathbb{Z}_R^m$ (vector of m elements and each element is in $[0, R)$)

3.2 Shamir's Secret Sharing

A t-out-of-n Shamir's secret sharing scheme (**SS**) [31] defined in a field $\mathbb{F}$, consists of two PPT algorithms:

- $\{(u, [s]_u)\}_{\forall u \in \mathcal{U}} \leftarrow$**SS.Share**$(s, t, \mathcal{U})$: splits a secret $s \in \mathbb{F}$ into n shares $[s]_u \in \mathbb{F}$, each of them for one user $u \in \mathcal{U}$. u are elements of the field $\mathbb{F}$ representing unique users, t is the reconstruction threshold, and n is the size of the users set $\mathcal{U}$. The algorithm first generates a polynomial $p(x)$ of uniformly random coefficients and of degree $t-1$ such that $p(0) = s$. It then computes $p(u) = [s]_u \forall u \in \mathcal{U}$.
- $s \leftarrow$**SS.Recon**$(\{(u, [s]_u)\}_{\forall u \in \mathcal{U}'}, t)$: reconstructs the secret $s \in \mathbb{F}$ from at least t shares. It is required that $\mathcal{U}' \subset \mathcal{U}$ and $|\mathcal{U}'| \geq t$. The algorithm uses the Lagrange interpolation [23] to compute the value of $p(0)$ as follows (all operation are in the field $\mathbb{F}$):

$$s = \sum_{\forall u \in \mathcal{U}'} \lambda_u [s]_u \qquad \lambda_u = \prod_{\forall v \in \mathcal{U}' \setminus \{u\}} \frac{v}{v-u}$$

3.3 Secret Sharing Over the Integers

We use a variant of Shamir's secret sharing which is defined over the integers (rather than in a field). The secret sharing scheme over the integers is defined by Rabin [30] and we denote it by **ISS**. The scheme shares an secret integer s in the interval $[-I, I]$ and provides σ-bits statistical security where σ is a security parameter. It is defined by the two PPT algorithms:

- $\{(u, [\Delta s]_u)\}_{\forall u \in \mathcal{U}} \leftarrow$**ISS.Share**$(s, t, \mathcal{U} = \{1, .., n\})$: splits a secret $s \in [-I, I]$ into n shares $[\Delta s]_u$, each of them for one user $u \in \mathcal{U}$ such that t is the reconstruction threshold. The algorithm first generates a polynomial $p(x)$ of uniformly random coefficients in $[-2^\sigma \Delta^2 I, 2^\sigma \Delta^2 I]$ and of degree $t-1$ such that $p(0) = \Delta s$ where $\Delta = n!$. It then computes $[\Delta s]_u = p(u)\ \forall u \in \mathcal{U}$.
- $s \leftarrow$**ISS.Recon**$(\{(u, [\Delta s]_u)\}_{\forall u \in \mathcal{U}'}, t)$: reconstructs the secret $s \in [-I, I]$ from at least t shares. It is required that $\mathcal{U}' \subset \mathcal{U}$ and $|\mathcal{U}'| \geq t$. The algorithm uses the Lagrange interpolation to compute the value of $p(0)$ as follows:

$$s = \frac{\sum_{\forall u \in \mathcal{U}'} \mu_u [\Delta s]_u}{\Delta^2} \qquad \mu_u = \frac{\Delta \cdot \prod_{v \in \mathcal{U}' \setminus \{u\}} (v)}{\prod_{v \in \mathcal{U}' \setminus \{u\}} (v-u)}$$

3.4 Key Agreement Scheme

We use the Diffie-Hellman key agreement scheme **KA**. It is parametrized with a security parameter λ consisting of three PPT algorithms:

- $pp = (p, q, g) \leftarrow$ **KA.Param**(λ): Given a security parameter λ it generates two large primes p and q such that q divides $p-1$ and outputs an element $g \in \mathbb{Z}_p^*$ of order q.
- $(c_u^{PK}, c_u^{SK}) \leftarrow$ **KA.Gen**(pp): This algorithm generate key pairs from public parameter where $(c_u^{PK}, c_u^{SK}) = (g^a, a)$ and $a \xleftarrow{R}$ [†] $\mathbb{Z}_q^*$.
- $c_{u,v} \leftarrow$**KA.Agree**$(pp, c_u^{SK}, c_v^{PK}, H)$: This algorithm uses the private key of user v, the public key of user u, and a hash function to generate a secret authentication and encryption key as $c_{u,v} = H((c_v^{PK})^{c_u^{SK}})$.

[†] This means chosen uiformally at random

3.5 Authenticated Encryption

An authenticated encryption scheme **AE** parametrized with a security parameter λ and a security key $k \in \{0,1\}^{\lambda}$ consists of two PPT algorithms:

- $c \leftarrow$**AE.Enc**(k,m): This algorithm uses the encryption key k to encrypt and authenticates a message m.
- $m \leftarrow$**AE.Dec**(k,c): This algorithm uses the same key to decrypt the ciphertext c verify its integrity.

3.6 Joye-Libert Secure Aggregation Scheme

This secure aggregation scheme proposed in [17]. The scheme involves the following parties: a trusted key-dealer, n users and the aggregator. Based on the formalized definition of secure aggregation (see Section 2), **JL** can be defined as following:

- $(sk_0, \{sk_u\}_{u\in\{1,..,n\}}, N, H) \leftarrow$ **JL.Setup**(λ): Given some security parameter λ, this algorithm generates two equal-size prime numbers p and q and sets $N = pq$. It randomly generates n secret keys $sk_u \xleftarrow{R} \pm\{0,1\}^{2l}$ where l is the number of bits of N and sets $sk_0 = -\sum_1^n sk_u$. Then, it defines a cryptographic hash function $H : \mathbb{Z} \rightarrow \mathbb{Z}^*_{N^2}$. It outputs the $n+1$ keys and the public parameters (N, H) .
- $y_{u,\tau} \leftarrow$ **JL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$: This algorithm encrypts private inputs $x_{u,\tau} \in \mathbb{Z}_N$ for time period τ using secret key $sk_u \in \mathbb{Z}_{N^2}$. It outputs cipher $y_{u,\tau}$ such that:

$$y_{u,\tau} = (1 + x_{u,\tau}N) \cdot H(\tau)^{sk_u} \mod N^2 \qquad (1)$$

- $X_\tau \leftarrow$ **JL.Agg**$(pp, sk_0, \tau, \{y_{u,\tau}\}_{u\in\{1,..,n\}})$: This algorithm aggregates the n ciphers received at time period τ to obtain $y_\tau = \prod_1^n y_{u,\tau}$ and decrypts the result. It obtains the sum of the private inputs $(X_\tau = \sum_1^n x_{u,\tau})$ as follows:

$$V_\tau = H(\tau)^{sk_0} \cdot y_\tau \qquad X_\tau = \frac{V_\tau - 1}{N} \mod N \qquad (2)$$

Theorem 3.1. *The scheme provides Aggregator Obliviousness security under the Decision Composite Residuosity (DCR) assumption [28] in the random oracle model.*

Proof. For the proof of this theorem, please refer to the original paper [17]. □

4 Fault-Tolerant Secure Aggregation using Masking

A previous solution from Bonawitz et al. [4] proposes a fault-tolerant version of secure aggregation. Authors build their protocol over a masking scheme. We first detail the basic masking scheme, we then show how Bonawitz et al. [4] create a fault-tolerant version of this scheme.

Secure Aggregation from Masking The clients first compute the masks using a Diffie-Hellman (DH) key agreement scheme (see Section 3.4). More specifically every client u generates a DH key pair (s_u^{SK}, s_u^{PK}) using **KA.Gen**. Then, each pair of users (u,v) agree on a shared key $s_{u,v}$ using **KA.Agree**. The user mask M_u is computed from the shared keys $\{s_{u,v}\}_{\forall v\in\mathcal{U}}$ as follows:

$$M_u = \sum_{\forall v\in\mathcal{U}:u<v} \mathbf{PRG}(s_{u,v}) - \sum_{\forall v\in\mathcal{U}:u>v} \mathbf{PRG}(s_{u,v})$$

Where **PRG** generates a vector whose size equals to the size of client's input. Note that the sum of the masks from all users is zero ($\sum_{\forall u\in\mathcal{U}} M_u = 0$). To protect the client input $X_u \in \mathbb{Z}_R^m$, the client u performs a one-time-pad encryption using the mask:

$$\mathbf{Protect}(pp, M_u, \perp, X_u) : Y_u = X_u + M_u \mod R$$

It can be shown easily that the aggregator can compute the sum of the clients' inputs by simply adding the masked inputs

$$\mathbf{Agg}(pp, \perp, \{Y_u\}_{\forall u\in\mathcal{U}}) : \sum_{\forall u\in\mathcal{U}} Y_u = \sum_{\forall u\in\mathcal{U}} X_u$$

Bonawitz et al. [4] Fault-Tolerant Solution The main problem of the previous solution is that it fails in case there are some dropping users, in particular the aggregator is not able to recover the sum of the masked inputs. To deal with this problem, Bonawitz et al. [4] proposed to integrate a t-out-of-n secret sharing [31]. In detail, each FL client shares with all the other clients its DH secret key. This way, if client u fails, a set of t or more clients can help the FL server to reconstruct the DH secret key s_u^{SK} of the missing client. Then, the server can recompute the mask M_u of the missing client and remove it from the aggregate. Additionally, this solution modifies the **Protect** algorithm by adding another masking layer using a blinding mask $B_u = \mathbf{PRG}(b_u)$ generated from a random generated number b_u.

$$\mathbf{Protect}(pp, M_u + B_u, \perp, X_u) : Y_u = X_u + M_u + B_u \mod R$$

The goal of this additional mask is to prevent the server (after reconstructing M_u) from revealing the clients input X_u. The value b_u is also shared using a t-out-of-n secret sharing. To aggregate the inputs, the server reconstructs b_u then compute B_u for all available clients ($\forall u \in \mathcal{U}_{on}$), and reconstruct s_v^{SK} then compute M_v for all failed clients ($\forall v \in \mathcal{U} \setminus \mathcal{U}_{on}$).

$$\mathbf{Agg}(\perp, \{Y_u\}_{\forall u\in\mathcal{U}_{on}}) : \sum_{\forall u\in\mathcal{U}_{on}} Y_u - \sum_{\forall u\in\mathcal{U}_{on}} B_u - \sum_{\forall u\in\mathcal{U}\setminus\mathcal{U}_{on}} M_u = \sum_{\forall u\in\mathcal{U}_{on}} X_u$$

5 Our Approach

From one-time mask to longer-term protection key . While the secure aggregation solution proposed in [4] is very

efficient during the masking of the individual inputs by the clients and the actual aggregation at the server, unfortunately this design incurs a significant computation cost at the beginning of each federated learning round whereby clients should run n DH key agreement protocol and call n PRGs to generate the mask of the actual FL round (n being the total number of clients). Indeed, the efficient masking scheme which consists of a simple addition with the mask is perfectly secure only if this mask is used once. Hence a random mask should be re-generated for each FL round. In our solution, we propose to replace the masking scheme with the **JL** secure aggregation scheme [17] (see section 3.6) whereby each client is given an individual secret key that can be used for several FL steps. The server's aggregation key will consist of the sum of all individual keys. While the protection algorithm at the client incurs more complex operations compared to the one in [4], clients do not need to perform a key agreement scheme at each FL round, anymore.

Secret sharing the clients keys A straightforward use of the **JL** scheme unfortunately does not address the problem of clients failures. Indeed, if the server does not receive at least one input from one client, the decryption of the aggregate result unfortunately fails. This is mainly due to the fact that the aggregation consists of multiplying all individually protected inputs which result in the protection of the aggregate result with the aggregation key. If one input misses, then the aggregate result becomes protected with a random key that the server does not know. Therefore, the server somehow needs to recover the protection of zeroes obtained with the individual keys of failed users. In order to cope with this problem, we propose to design an threshold-variant of the **JL** scheme whereby clients can secret share their individual keys with other clients so that when one client fail, t out of n clients that are still online help provide a protected zero-value that is computed with the failed user's individual key so that the server can correctly compute the aggregated result.

Masking against malicious behavior Similar to [4], with the use of the threshold-variant of **JL** scheme which help recover from the event of failed clients, the server can maliciously claim that a certain client failed while it actually is online. Such a malicious behavior can result in the leakage of the client's individual input. Indeed, by receiving the protection of two different values with the same key for the same time period, the **JL** scheme can leak this value to the server (see the definition of aggregator obliviousness in Section 2). To cope with such behavior, we propose to follow the same approach proposed in [4]: Each input is blinded with a mask which is generated with a pseudo-random generator whose input seed is secretly shared among all clients. Thanks to this additional blinding, the correct aggregation of the result also relies on the collaboration of honest clients: Indeed, either the server claims that one user is offline and then t out of n clients help recover the zero-value protected with the failed client's key or this user is not claimed offline and the server receives the shares of its blinding mask. Honest users will never perform both operations for a given user at the same time.

6 Fault-Tolerant Secure Aggregation using TJL

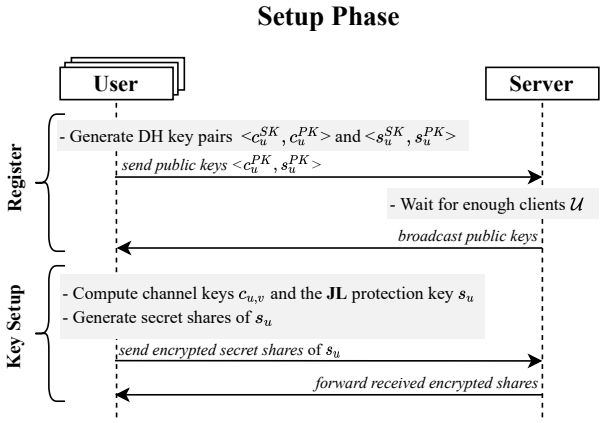


Figure 2: Overview of the setup phase of the secure and fault-tolerant aggregation protocol

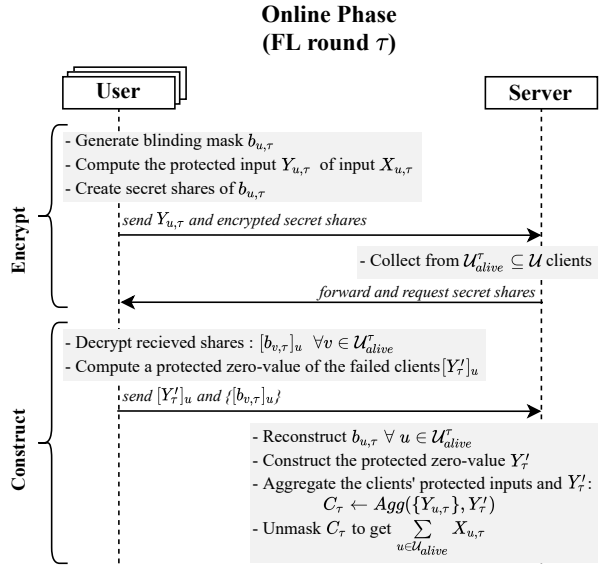


Figure 3: Overview of the online phase of the secure and fault-tolerant aggregation protocol

In this section, we describe the newly designed secure and

fault-tolerant aggregation protocol.

The protocol consists of a setup phases and an online phase. In the setup phase (see Figure 2), two rounds of communication are performed: the first whereby clients register by sharing their public keys and receive the public parameters; and the second whereby clients shares and receives shares of the **JL** protection keys. In the online phase (see Figure 3), also two round of communication are performed: the first whereby the clients send their protected inputs to the server and share their blinding mask with other clients; and the second whereby the clients allow the aggregator to reconstruct the aggregate by sending the shares of the protected zero-values of failed clients and the shares of the blinding mask of available ones. The complete specification of the entire protocol is described in Figure 5. In the sequel of this section, we first design the threshold-variant of the **JL** scheme that can also be considered as an independent building block; we further explain how to extend this scheme for the use of vectors instead of single values by describing a dedicated encoding scheme; We finally describe how actually keys are distributed and the computation of the resulting aggregation key.

6.1 Threshold-variant of the Joye-Libert secure aggregation scheme

In this section, we describe a *threshold-variant* of the Joye-Libert secure aggregation scheme (see section 3.6 for the original scheme). The design of this scheme mainly transplants the design of the threshold variant of the Paillier encryption scheme [7] into this context. This extended solution will mainly help the server recover failed users' inputs (which consist of the protection of the zero-value under each failed user's individual key) and hence compute the final aggregate value. Hence, the goal is to distribute a user key sk_u to the n users such that any subset of at least t (online) users can produce a ciphertext on behalf of user u while less than t users have no useful information. The main building block to distribute such a key is a secret sharing scheme where inputs are in $\{0,1\}^{2l}$ and l corresponds to the bit-size of the modulus N. Hence, the threshold-variant Joye-Libert secure aggregation scheme, denoted as **TJL**, would consist of the following PPT algorithms:

- $(sk_0, \{sk_u\}_{u\in\{1,..,n\}}, N, H, \sigma) \leftarrow$ **TJL.Setup**(λ): Given some security parameter λ, this algorithm basically calls the original **JL.Setup**(λ) algorithm and outputs the server key, one secret key per user and the public parameters. Additionally, it chooses the security parameter of the **ISS** scheme σ.
- $\{(v, [\Delta sk_u]_v)\}_{\forall v\in\mathcal{U}} \leftarrow$ **TJL.SKShare**$(sk_u, t, \mathcal{U})$: On input of user u's secret key, this algorithm calls **ISS.Share**$(sk_u, t, \mathcal{U})$ (see Section 3.3) where the interval of the secret sk_u is $[-2^{2l}, 2^{2l}]$ and l is the number of bits of the modulus N. Indeed, similar to [27], in our solution, we construct a secret sharing of the private key sk_u over the integers. Hence, this algorithm outputs n shares of user u's key sk_u, each share $[\Delta sk_u]_v$ is for each user $v \in \mathcal{U}$.
- $[y'_\tau]_u \leftarrow$ **TJL.ShareProtect**$(pp, \{[\Delta sk_v]_u\}_{v\in\mathcal{U}''}, \tau)$: This algorithm protects a zero-value with user u's shares of all the secret keys corresponding to the failed users ($v \in \mathcal{U}''$) ($[\Delta sk_v]_u$ is the user u's share of the secret key sk_v corresponding to the failed user v). It basically calls **JL.Protect**$(pp, \sum_{v\in\mathcal{U}''}[\Delta sk_v]_u, \tau, 0)$ and outputs $[y'_\tau]_u = H(\tau)^{\sum_{v\in\mathcal{U}''}[\Delta sk_v]_u} \mod N^2$. This algorithm is called when there are failed users and hence their input need to be recovered.
- $y'_\tau \leftarrow$ **TJL.ShareCombine**$(\{(u, [y'_\tau]_u, n)\}_{\forall u\in\mathcal{U}', t})$: This algorithm combines t out of n protected shares of the zero-value for time step τ and given $\Delta = n!$. $\mathcal{U}'$ is a subset of the online users such that $|\mathcal{U}'| \geq t$ and $\mathcal{U}''$ is the set of failed users. Similar to the solution in [27], it computes the Lagrange interpolation on the exponent (the μ_u coefficients are defined in **ISS.Recon** in Section 3.3):

$$\begin{aligned} y'_\tau = \prod_{\forall u\in\mathcal{U}'} ([y'_\tau]_u)^{\mu_u} &= H(\tau)^{\sum_{u\in\mathcal{U}'}\mu_u\sum_{v\in\mathcal{U}''}[\Delta sk_v]_u} \\ &= H(\tau)^{\sum_{v\in\mathcal{U}''}\sum_{u\in\mathcal{U}'}\mu_u[\Delta sk_v]_u} \\ &= H(\tau)^{\Delta^2\sum_{v\in\mathcal{U}''}sk_v} \end{aligned}$$

- $y_{u,\tau} \leftarrow$ **TJL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$: This algorithm mainly calls **JL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$ and hence outputs the cipher $y_{u,\tau}$. This algorithm is mainly used by all users to protect their input before the aggregator collects them.
- $X_\tau \leftarrow$ **TJL.Agg**$(pp, sk_0, \tau, \{y_{u,\tau}\}_{\forall u\in\mathcal{U}'}, y'_\tau)$: On input the public parameters pp, the aggregation key sk_0, the individual ciphertexts of online users ($u \in \mathcal{U}'$), and the ciphertexts of the zero-value corresponding to the failed users, this algorithm aggregates the ciphers of time period τ by first multiplying the inputs for all online users, raising them to the power Δ^2, and multiplying the result with the ciphertext of the zero-value. $\mathcal{U}'$ is that set of online users and $\mathcal{U}'' = \mathcal{U} \setminus \mathcal{U}'$ is the set of failed users. It computes:

$$\begin{aligned} y'_\tau &= (\prod_{\forall u\in\mathcal{U}'} y_{u,\tau})^{\Delta^2} \cdot y'_\tau \mod N^2 \\ &= (1+\Delta^2 \sum_{\forall u\in\mathcal{U}'} x_{u,\tau}N) H(\tau)^{\Delta^2 \sum_{\forall u\in\mathcal{U}'} sk_u} \cdot H(\tau)^{\Delta^2 \sum_{\forall u\in\mathcal{U}''} sk_u} \\ &= (1+\Delta^2 \sum_{\forall u\in\mathcal{U}'} x_{u,\tau}N) H(\tau)^{\Delta^2 \sum_{\forall u\in\mathcal{U}} sk_u} \\ &= (1+\Delta^2 \sum_{\forall u\in\mathcal{U}'} x_{u,\tau}N) H(\tau)^{-\Delta^2 sk_0} \end{aligned}$$

To decrypt the final result, the algorithm proceeds as

follows:

$$V_\tau = H(\tau)^{\Delta^2 sk_0} \cdot y'_\tau \qquad X_\tau = \frac{V_\tau - 1}{N\Delta^2} \mod N \quad (3)$$

Theorem 6.1. *This scheme provides Aggregator Obliviousness security under the DCR assumption in the random oracle model if the number of corrupted users is less than the threshold t.*

Proof. The security of this scheme mainly relies on the security of the JL secure aggregation scheme which is proved secure under the DCR assumption 3.1 and the security of the secret sharing scheme over integers which is also proved to be statistically secure in Theorem 1 in [38]. □

6.2 Input vector encoding

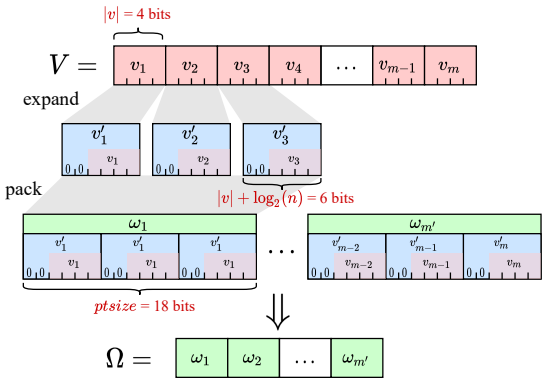


Figure 4: Illustration of the vector encoding scheme. In this example, the plain-text size is $ptsize = 18$ bits and the number of supported additions $n = 4$. The vector V has elements v of size $s = 4$ bits. First v is expanded by $\log_2(n) = 2$ bits. Then, each $\lfloor \frac{ptsize}{s+\log_2(n)} \rfloor = 3$ values are packed to one long integer ω. The final encoded vector Ω is composed of all the long integers. The compression ration is $cr = \frac{|V|}{|\Omega|} = \frac{m}{m'} = 3$

The **TJL** scheme is originally designed to work with single value integers that are less than N. In the case of federated learning applications, FL clients usually send a vector of parameters instead of a single one. A naïve solution to cope with vectors of several elements would be to independently call the **TJL** protection and aggregation algorithms for each element of the vector. Such a design choice would nevertheless incur a significant overhead mainly because each protected element's size is equal to the size of the key.

We therefore propose a dedicated encoding solution that compresses the vector of m inputs into a smaller vector of m' integers while packing multiple elements in one big integer. Each element of the initial vector V is firstly expanded by $\log_2(n)$ bits of 0's at the beginning of the element. Then several elements of the vector are packed to form a large integer ω. The number of elements that ω can represent corresponds to the $ptsize$ which denotes the plaintext size of the **TJL** scheme divided by the actual size of the extended element (i.e $\lfloor \frac{ptsize}{s+\log_2(n)} \rfloor$, s being the bit length of a vector element). Hence vector $\Omega = [\omega_1, ..., \omega_{m'}]$ becomes a compressed encoded vector of V and the compression ratio is equal to $cr = \frac{|V|}{|\Omega|} = \lfloor \frac{ptsize}{s+\log_2(n)} \rfloor$. Note that for **TJL** scheme, the plaintext size is equal to half the size of the user key ($ptsize = \frac{|sk_u|}{2}$) and thus the compression ratio becomes $cr = \lfloor \frac{|sk_u|}{2(s+\log_2(n))} \rfloor$. Figure 4 illustrates the compression operation.

The decoding operation can be simply computed by unpacking ω_i into bitmaps of $s + \log_2(n)$.

To evaluate **TJL.Protect** and **TJL.ShareProtect** algorithms on vectors, the user input and respectively the vector of zeros are first encoded. Then, the algorithm are applied on each element of the encoded vector. We use the index of the element i to generate a unique time period for each element in the vector. For example, to protect $X_{u,\tau}$, we run **TJL.Protect**$(sk_u, \tau || i, X'_{u,\tau}[i])$ where $X'_{u,\tau}[i]$ is the i^{th} element of the vector $X'_{u,\tau}$ of $X_{u,\tau}$. Similarly, **TJL.ShareCombine** and **TJL.Agg** are also applied on each element in the vector.

6.3 Distributing the Keys of JL Scheme

Both the original JL scheme and the new **TJL** scheme require the existence of a trusted key dealer for the execution of the **TJL.Setup** algorithm. However, due to the decentralized nature of FL applications the dependence on a third party key dealer should be avoided. We therefore propose to follow the same approach as in [4] and design a decentralized **TJL.Setup** algorithm whereby the aggregator key sk_0 becomes 0 and user keys are defined as follows: each pair of clients (u, v) agree using the **KA** scheme on a shared mutual key $sk_{u,v}$; Then, user u computes its protection key sk_u as follows: $sk_u \leftarrow \sum_{v \in \mathcal{U}} (\delta_{u,v} \cdot sk_{u,v})$ where $\delta_{u,v} = 1$ when $u > v$, and $\delta_{u,v} = -1$ when $u < v$.

The correctness of the **TJL** aggregation protocol is preserved since:

$$\sum_{\forall u \in \mathcal{U}} sk_u = \sum_{\forall u \in \mathcal{U}} \Big(\sum_{\forall v \in \mathcal{U}} \delta_{u,v} \cdot sk_{u,v} \Big) = 0 = -sk_0$$

Furthermore, the security proof in [17] considers the case where an adversary controls the key of the aggregator. In such case, the scheme cannot protect the result of the aggregation but it still protects the individual inputs of the users which is sufficient for FL. Note that this decentralized solution allows the distribution of the **TJL** user keys but it does not completely release the dependency on a trusted third party to generate the public parameters. There are some techniques to distribute the computation of the public modulus N presented in [27, 38]. In this work, we assume an offline trusted third party that distributes the public parameters only.*

*We consider the problem of full independence on a trusted party as part of some future work.

Secure Aggregation Protocol - Setup Phase

- **Setup - Register:**
  Trusted Dealer:
  - Choose security parameters λ and runs $pp^{KA} \leftarrow$ **KA.Param**(λ) and $(\perp,\perp,N,H,\sigma) \leftarrow$ **TJL.Setup**(λ). It sets the public parameters $pp = (pp^{KA},N,H,\sigma,t,n,m,R,\mathbb{F})$ such that t is the secret sharing threshold, n is the number of clients, $\mathbb{Z}_R^m$ is the space from which inputs are sampled, and $\mathbb{F}$ is the field for **SS** scheme. It sends them to the server and to all the clients.

  User u (pp):
  - Receive the public parameters from the trusted dealer.
  - Generate key pairs $(c_u^{PK},c_u^{SK}) \leftarrow$ **KA.gen**$(pp^{KA}),(s_u^{PK},s_u^{SK}) \leftarrow$ **KA.gen**(pp^{KA})
  - Send $(c_u^{PK} \,||\, s_u^{PK})$ to the server (through the private authenticated channel) and move to next round.

  Server(pp):
  - Receives public parameters pp from the trusted dealer.
  - Collect all public keys from the users (denote with $\mathcal{U}$ the set of registered users). Abort if $|\mathcal{U}| < t$ otherwise move to the next round.
  - Broadcast to all users the list $\{u,(c_u^{PK},s_u^{PK})\}_{\forall u \in \mathcal{U}}$
- **Setup - Key Setup:**
  User u:
  - Receive the public keys of all registered users $\mathcal{U}$
  - For each registered user $v \in \mathcal{U} \setminus \{u\}$, compute channel keys $c_{u,v} \leftarrow$ **KA.agree**$(pp^{KA},c_u^{SK},c_v^{PK},H)$.
  - For each registered user $v \in \mathcal{U} \setminus \{u\}$, compute $sk_{u,v} \leftarrow$ **KA.agree**$(pp^{KA},s_u^{SK},s_v^{PK},H^*)$ (set $sk_{u,u}=0$). Then compute the **TJL** secret key $s_u \leftarrow \sum_{v \in \mathcal{U}} \delta_{u,v} \cdot sk_{u,v}$ where $\delta_{u,v}=1$ when $u>v$, and $\delta_{u,v}=-1$ when $u<v$.
  - Generate t-out-of-$|\mathcal{U}|$ shares of the **TJL** secret key: $\{(v,[sk_u]_v)\}_{\forall v \in \mathcal{U}} \leftarrow$ **TJL.SKShare**$(sk_u,t,\mathcal{U})$.
  - For each registered user $v \in \mathcal{U} \setminus \{u\}$, encrypt its corresponding shares: $\varepsilon_{u,v} \leftarrow$ **AE.enc**$(c_{u,v},u \,||\, v \,||\, [sk_u]_v)$.
  - If any of the above operations fails, abort.
  - Send all the encrypted shares $\{\varepsilon_{u,v}\}_{\forall v \in \mathcal{U}}$ to the server (each implicitly containing addressing information u,v as metadata).
  - Store all messages received and values generated in the setup phase, and move to the online phase.

  Server:
  - Collect from each user u its encrypted shares $\{(u,v,\varepsilon_{u,v})\}_{\forall v \in \mathcal{U}}$.
  - Forward to each user $v \in \mathcal{U}$ its corresponding encrypted shares: $\{(u,v,\varepsilon_{u,v})\}_{\forall u \in \mathcal{U}}$ and move to the online phase.

Secure Aggregation Protocol - Online Phase

- **Online - Encrypt (step τ):**
  User u:
  - Sample a random element $b_{u,\tau} \xleftarrow{R} \mathbb{F}$ (to be used as a seed for a **PRG**).
  - Extend $b_{u,\tau}$ using the **PRG**: $B_{u,\tau} \leftarrow PRG(b_{u,\tau})$.
  - Protect the private input $X_{u,\tau} \in \mathbb{Z}_R^m$ (after masking it with $B_{u,\tau}$) using **TJL** scheme: $Y_{u,\tau} \leftarrow$ **TJL.Protect**$(pp,sk_u,\tau,X_{u,\tau}+B_{u,\tau})$.
  - Generate t-out-of-$|\mathcal{U}|$ shares of $b_{u,\tau}$ using the **SS** scheme: $\{(v,[b_{u,\tau}]_v)\}_{\forall v \in \mathcal{U}} \leftarrow$ **SS.Share**$(b_{u,\tau},t,\mathcal{U})$.
  - For each registered user $v \in \mathcal{U} \setminus \{u\}$, encrypt its corresponding shares $e_{(u,v),\tau} \leftarrow$ **AE.Enc**$(c_{u,v},u \,||\, v \,||\, [b_{u,\tau}]_v)$
  - If any of the above operations fails, abort.
  - Send all the encrypted shares $\{e_{(u,v),\tau}\}_{\forall v \in \mathcal{U}}$ (with addressing information u,v as metadata) and the protected input $Y_{u,\tau}$ to the server .

  Server:
  - Collect from each user u its encrypted shares $\{(u,v,e_{(u,v),\tau})\}_{\forall v \in \mathcal{U}}$ and its protected input $Y_{u,\tau}$ (or time out).
  - Denote with $\mathcal{U}_{on}^{\tau} \subset \mathcal{U}$ the set of online users. Abort if $|\mathcal{U}_{on}^{\tau}| < t$.
  - Forward to each user $v \in \mathcal{U}_{on}^{\tau}$ its corresponding encrypted shares: $\{(u,v,e_{(u,v),\tau})\}_{\forall u \in \mathcal{U}_{on}^{\tau}}$.
- **Online - Construct (step τ):**
  User u:
  - Receive the encrypted shares and deduce the list of online users $\mathcal{U}_{on}^{\tau}$ from the received shares . Verify that $\mathcal{U}_{on}^{\tau} \subset \mathcal{U}$ and $|\mathcal{U}_{on}^{\tau}| >= t$.
  - Decrypt all the encrypted secret shares: $v' \,||\, u' \,||\, [b_{v,\tau}]_u \leftarrow$ **AE.Dec**$(c_{u,v},e_{(v,u),\tau})$. Assert that $u=u' \wedge v=v'$
  - Compute the share of the zero-value corresponding to all failed users: $[Y'_\tau]_u \leftarrow$ **TJL.ShareProtect**$(pp,\{[s_v]_u\}_{\forall v \in \mathcal{U} \setminus \mathcal{U}_{on}^{\tau}},\tau)$.
  - Abort if any operation failed.
  - Send the secret shares of the blinding mask seeds $\{[b_{v,\tau}]_u\}_{\forall v \in \mathcal{U}_{on}^{\tau}}$ and of the share of the protected zero-value $[Y'_\tau]_u$ to the server.

  Server:
  - Collect shares from at least t honest users. Denote with $\mathcal{U}_{shares}^{\tau} \subset \mathcal{U}_{on}^{\tau}$ the set of users. Abort if $|\mathcal{U}_{shares}^{\tau}| < t$.
  - Construct the blinding mask seed of all users $b_{u,\tau} \forall u \in \mathcal{U}_{on}^{\tau} : b_{u,\tau} \leftarrow$ **SS.Recon**$(\{[b_{u,\tau}]_v\}_{\forall v \in \mathcal{U}_{shares}^{\tau}},t)$
  - Recompute the blinding mask: $B_{u,\tau} \leftarrow PRG(b_{u,\tau})$
  - Construct the protected zero-value corresponding to all failed users: $Y'_\tau \leftarrow$ **TJL.ShareCombine**$(\{[Y'_\tau]_v\}_{\forall v \in \mathcal{U}_{shares}^{\tau}},t)$
  - Aggregate all the protected inputs of the online clients and the protected zero-value: $C_\tau \leftarrow$ **TJL.Agg**$(pp,0,\tau,\{Y_{u,\tau}\}_{\forall u \in \mathcal{U}_{on}^{\tau}},Y'_\tau\})$
  - Remove the blinding masks $C_\tau - \sum_{\forall u \in \mathcal{U}_{on}^{\tau}} B_{u,\tau} = \sum_{\forall u \in \mathcal{U}_{on}^{\tau}} X_{u,\tau}$

Figure 5: Detailed description of the secure and fault-tolerant aggregation protocol

*A Full Domain Hash function using the hash function H to map the **KA** shared key to a **JL** secret key

Table 1: Complexity analysis of the online phase of both protocols (our protocol vs [4]). n is the number of clients and m is the dimension of the client's input.

		Client	Server
Computation	*CCS17* [4]	$O(n^2+nm)$	$O(n^2m)$
	Ours	$O(n^2+m)$	$O(n^2+nm)$
Communication	*CCS17* [4]	$O(n+m)$	$O(n^2+nm)$
	Ours	$O(n+m)$	$O(n^2+nm)$
Storage	*CCS17* [4]	$O(n+m)$	$O(n^2+m)$
	Ours	$O(n+m)$	$O(n^2+nm)$

7 Performance Evaluation

In this section, we analyze ana evaluate the scalability of our solution in terms of the computation, communication, and storage at both the client and the server. We perform the complexity analysis with respect the number of clients n and the dimension of the client's input m. We conduct a comparative analysis with the solution in [4] (see Table 1). We only present the analysis of the online phase of both protocols.

7.1 Client Performance

Communication $O(n+m)$. The communication cost consists of: In round *Encrypt*, (1) sending $(n-d-1)$ shares of $b_{u,\tau}$ and receiving $(n-d-1)$ shares, and (2) sending the encrypted client input of size $\lceil \frac{ma_k}{cr} \rceil$ ($cr = \frac{a_k}{2(s+\log_2(n))}$ is the compression ratio, where s is the bit length of each element in clients input and a_k is the JL key size, see Section 6.2). In round *Construct*, (3) sending $(n-d-1)$ shares of $b_{u,\tau}$, , and (4) if at least one client failed, sending the share of the protected the zero-value $Y'_{u,\tau}$ of size $\lceil \frac{ma_k}{cr} \rceil$. In total, $(3n-3d-3)a_s + 2(1+p)m(l+\lceil \log_2(n) \rceil)$ where a_s is the bit length of a secret share and p is the probability that at least one client failed). In Bonawitz et al. scheme [4] , the total communication cost is: $2na'_k + (5n-4)a_s + m(l+\lceil \log_2(n) \rceil)$ where a'_k is the bit length of the public key in a key exchange. The relative scalability of our protocol compared to [4] with respect to the number of clients and the clients dimension are respectively $\frac{2a'_k+5a_s}{3a_s}$ and $\frac{1}{2(1+p)}$. Choosing $a'_k = 256, a'_s = 128$, our protocol scales 3 times **better** with respect to the number of clients and 4 times **worse** with respect to dimension of the input if some clients fail in each FL step ($p = 1$). For example, for $n = 2^{12}, m = 2^{14}, s = 16$: CCS17 incurs communication cost of 0.63MB while ours 0.35 MB (0.3 MB without client failures).

Computation $O(n^2+m)$. The computation cost consists of: In round *Encrypt*, (1) the generation of t out of n shares of $b_{u,\tau}$ which is $O(n^2)$, and (2) the encryption of the client's message $X_{u,\tau}$ which is $O(m)$. In round *Construct*, (3) the encryption of the zero-value using the secret shares which is $O(m)$. Therefore, the total complexity is $O(n^2+m)$ which is **lower** than in [4] $O(n^2+nm)$. The higher complexity in [4] is due to the one-to-one key agreement on the shared masking seeds then extending each of them to the dimension of the client's input which add an extra $O(nm)$.

Storage $O(n+m)$. The user must store the keys and shares of each other user as well as the data vector which results in a storage overhead of $O(n+m)$ which is the **same** as the one in [4].

7.2 Server Performance

Communication $O(n^2+nm)$. The only message exchanges happening in the protocol is between the server and the clients. Therefore, the server's communication cost is n times the client's communication cost. Thus, a complexity of $O(n^2+nm)$ which is the **same** as the one in [4].

Computation $O(n^2+nm)$. The server's computation cost consists of: In round *Encrypt*, (1) computing the product of the received ciphertexts which is $O((n-f)m)$ where f is the number of failed clients. In round *Construct*, (2) reconstructing t out of n shares of $b_{u,\tau}$ for each online user u which is $O((n-f)^2)$, (3) extending the seed $b_{u,\tau}$ to the dimension of the client's input (using **PRG**) which us $O((n-f)m)$, if at least one client failed, (4) constructing the protected zero-value Y'_τ from its t shares which is $O((n-f)m)$, and (5) aggregating the ciphertexts and unmasking the result which is $O((n-f)m)$. The total computation cost equals to $O((n-f)^2+(n-f)m) \equiv O(n^2+nm)$ which is **lower** than the one in [4] $O(n^2+d(n-f)m) \equiv O(n^2m)$. The higher complexity in [4] is because the server should recompute $n-f$ key agreements for each failed client then extend each of them to the dimension of the client's input which is $O(f(n-f)m)$. This analysis shows that our protocol scales **better** when the number of failures increase.

Note that the reconstruction of t out of n shares normally costs $O(n^2)$ since it consists of computing the Lagrange coefficient $O(n^2)$ then applying the Lagrange formula $O(n)$. So, the computation of t out of n shares for n users should cost $O(n^3)$. However, for both protocols we optimize the reconstruction by computing the Lagrange coefficients only one time per FL round and use them for all the reconstructions which results in $O(n^2+n) \equiv O(n^2)$.

Storage $O(n^2+nm)$. The server storage consists of: t shares of $b_{u,\tau}$ for each client $b_{u,\tau}$ which is $O(n^2)$ and t shares of Y'_τ which is $O(nm)$. Thus, a total of $O(n^2+nm)$ which is **higher** than the one in [4] $O(n^2+m)$.

# Clients	% Failures			
	0%	10%	20%	30%
100	9.8 sec	20.9 sec	20.8 sec	20.8 sec
300	10.9 sec	28.1 sec	27.9 sec	26.6 sec
600	10.5 sec	35.5 sec	35.6 sec	35.8 sec

Table 2: Wall-clock running time per client in the online phase for one FL round with different % of client failures. The number of clients varies $n = \{100, 300, 600\}$ and the dimension is fixed to $m = 100000$.

8 Experimental Evaluation

We have implemented a prototype of our protocol and a prototype of the protocol presented in [4] using Python programing language and conducted an experimental evaluation. We plan to share both prototypes with the public. We first discuss the implementation details of the various cryptographic primitives. Then, we study the computation time and the bandwidth cost for the FL clients and serve. The measurements were performed using a single threaded process on a machine with a Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz processor and 32 GB of RAM.

8.1 Implementation Details

We use the same implementation and parameters for the building blocks of our protocol and the protocol proposed in [4].

- *Pseudo-Random Generator (**PRG**)*: We use AES encryption [12] in the counter mode with 128 bits key size. Thus, the blinding mask seed (b_u) is 128-bit long.
- *Key Agreement (**KA**)*: We use Elliptic-Curve Diffie-Hellman over the NIST P-256 curve. For the hash function we use SHA256.
- *Secret Sharing (**SS**)*: We use the finite fields $\mathbb{F}_p$ (integers modulo p where p is prime) in the implementation of Shamir's secret sharing. To share the seed of the blinding mask (b_u) (128 bits) we set $p = 2^{129} - 1365$. To share the DH secret key (256 bits) we set $p = 2^{257} - 2233$.
- *Authenticated Encryption (**AE**)*: We use AES-GCM [13] with a key size of 256 bits.
- *Threshold Joye-Libert Scheme (**TJL**)*: We use 1024 bits for the public parameter N. Thus, the user keys are of size 2048 bits. For the hash function $H : \mathbb{Z} \to \mathbb{Z}^*_{N^2}$, we implement it as a Full-Domain Hash using a chain of eight SHA256 hashes. Additionally, for the secret sharing over the integers scheme **ISS** used by **TJL.SKShare**, we set the security parameter σ to 128 bits.

# FL rounds	Total Time	Setup Time
100	451.3 sec	9.6 sec (2.1%)
500	2218.3 sec	9.6 sec (0.4%)
1000	4427.0 sec	9.6 sec (0.2%)

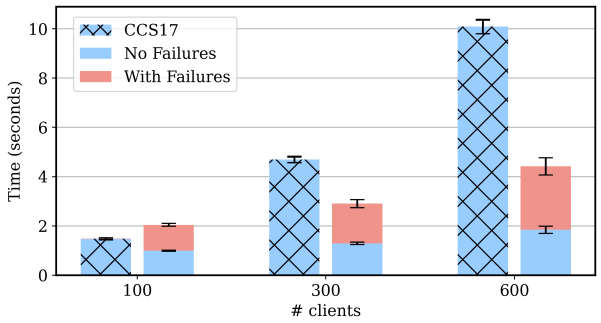
Table 3: Wall-clock running time per client for $T = \{100, 500, 1000\}$ FL rounds. The number of clients, dimension and % of failures are fixed to $n = 600, m = 100000, f = 10\%$. Total time is the $setup\_time + T \times online\_time$.

8.2 Several Experiments with Measurements

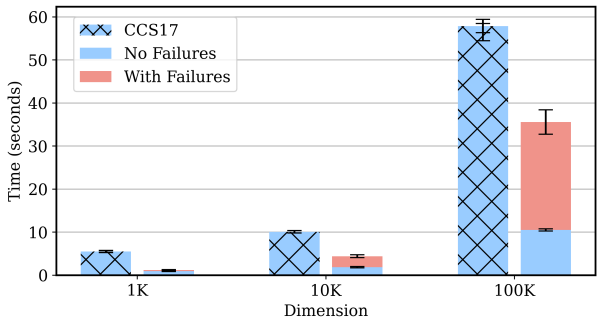
We run several experiments with our protocol and the one in [4] protocol (CCS17) while varying the number of the clients $n = \{100, 300, 600\}$. We also use different dimensions for the clients inputs $m = \{1000, 10000, 100000\}$ and different client failure percentages $f = \{0\%, 10\%, 30\%\}$. Failures happen before the client sends its protected input.

Running time for clients We plot the wall-clock running time for the clients in both protocols (i.e., our protocol and the one in [4]) in Figure 6. Our protocol shows better running time in most of the measured scenarios. Additionally, Figure 6a shows that our protocol scales better with respect to the increasing number of clients (i.e., our solution is $\times 2.7$ faster with 100 clients and $\times 4.6$ faster with 600). This confirms the study in Section 7.1. On the other hand, when increasing the dimension of the client's input, our protocol results in a larger increase in the running time (see Figure 6b). This is mainly due to the calls to **TJL.Protect** and **TJL.ShareProtect** algorithms that have a higher cost than evaluating a **PRG** as the case in [4]. Moreover, our protocol has an additional overhead when client failures occur. However, this overhead remains constant w.r.t. the percentage of client failures (see Table 2). This is expected since a user u computes a single value Y'_τ for all failed clients. We also show the setup time for our protocol in Table 3. The results show that the offline time becomes negligible after running a sufficient number of rounds of the FL protocol.

Running time for the server We plot the wall-clock running time for the server in both protocols (i.e., our protocol and the one in [4]) in Figure 7. The results show that clients' failures significantly affect the performance of the server for both protocols. This is because the server should run a heavy reconstruction phase. More importantly, our protocol shows better performance in the cases of many failures (30% of the clients failed) but lower performance in the cases of few failures (10% of the clients failed). This is because the reconstruction overhead at the server in [4] increases with the number of failed clients (i.e., computation of DH key agreements for each failed client). Furthermore, in our protocol the reconstruction overhead is constant w.r.t. the number of failed

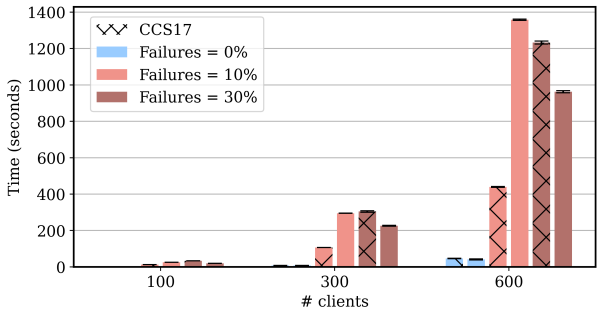


(a) Varying the number of clients $n = \{100, 300, 600\}$, the input dimension is fixed to $m = 10000$.

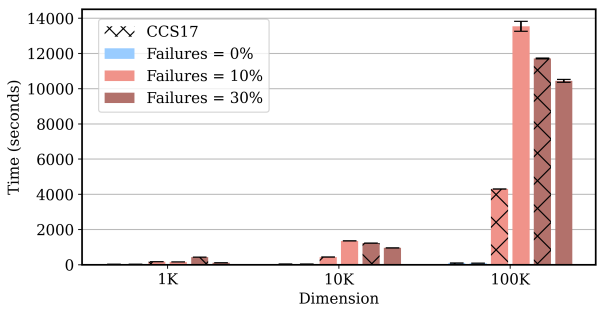


(b) Varying the dimension of clients' inputs $m = \{1000, 10000, 100000\}$, # clients is fixed to $n = 600$.

Figure 6: Wall-clock running time per client. The measurements are performed using a single-threaded python implementation of our solution and the solution in [4] (only the online phase time is shown). Measurements are taken in two scenarios, without client failures and with 10% failed clients. The values represent the average wall-clock time of all running clients. The error bars represent the standard deviation.



(a) Varying the number of clients $n = \{100, 300, 600\}$, the input dimension is fixed to $m = 10000$.



(b) Varying the dimension of clients' inputs $m = \{1000, 10000, 100000\}$, # clients is fixed to $n = 600$.

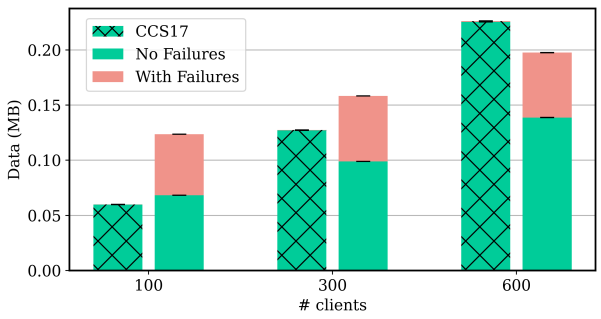
Figure 7: Wall-clock running time for the server. The measurements are performed using a single-threaded python implementation of our solution and the solution in [4] (only the online phase time is shown). Measurements are repeated with different client failures $f = \{0\%, 10\%, 30\%\}$. The values represent the average wall-clock time of 5 end-to-end runs. The error bars represent the standard deviation

clients (i.e., construction of single value Y'_τ that represents all failed clients). Interestingly, when more clients fail, the number of online users decrease which reduces the number of reconstructed $b_{u,\tau}$ (see Section 7.2). This results in a decrease in the server running time for our protocol as the number of failed clients increase. We show the detailed running time (per protocol round) in Table 4 in the Appendix. Notice that the running time for both protocols can be significantly reduced by implementing them using a pre-compiled programming language (Eg., Java). Indeed, a comparison between our implementation of [4] and the implementation reported in the original paper shows a $\times 50$ faster execution.

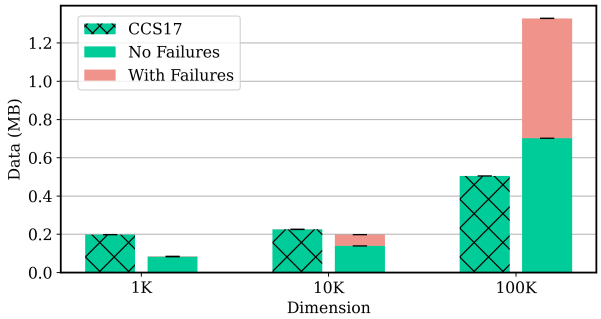
Data transfer We plot the total data transfer (sent and received) per client in both protocols (i.e., our protocol and the protocol in [4]) in Figure 8 (the measurements for the server are essentially the same as for the client multiplied by the number of client). When running with clients input of dimension $m = 10000$, both protocols show bandwidth cost less than 250 KB per client. Our protocol has larger data transfer in scenarios with low number of client ($n = 100$) but is more efficient when the number of clients is increased to $n = 600$. On the other hand, when working with larger dimensions for clients input, our protocol has larger data transfer. This confirms the analysis in Section 7.1. Moreover, our protocol incurs additional overhead in case of failed client. This overhead is constant w.r.t. the percentage of failed clients and it is equal to the size of Y'_τ. We show detailed measurements per protocol round in Table 5 in the Appendix.

9 Security Analysis

In this section, we evaluate the security of our secure and fault-tolerant aggregation protocol and prove that it ensures *Aggregator Obliviousness* (AO) in both the honest-but-curious and active adversary setting (see Section 2) given a dedicated threshold t of honest clients.



(a) Varying the number of clients $n = \{100, 300, 600\}$, the input dimension is fixed to $m = 10000$.



(b) Varying the dimension of clients' inputs $m = \{1000, 10000, 100000\}$, # clients is fixed to $n = 600$.

Figure 8: Total data transfer (sent and received) per client. The measurements are performed using a single-threaded python implementation of our solution and the solution in [4] (only the online phase time is shown). Measurements are taken in two scenarios, without client failures and with 10% failed clients. The values represent the average data sent and received of all running clients. The error bars represent the standard deviation.

Security in the honest-but-curious model In the honest-but-curious model, we assume that the aggregator correctly follows the protocol but it colludes with up to $n-t$ clients. Let $\mathcal{U}_{corr}$ be the set of corrupted clients and $\mathcal{C} = \mathcal{U}_{corr} \cup \mathcal{S}$ ($\mathcal{S}$ represents the server). The view of $\mathcal{C}$ is computationally indistinguishable from a simulated view if the number of corrupted clients is less than the threshold t ($|\mathcal{U}_{corr}| < t$). Based on that, the minimum number of honest clients t should be strictly larger than half of the number of clients in the protocol ($t > \frac{n}{2}$). Hence the protocol can recover from up to $\frac{n}{2} - 1$ client failures.

To prove this argument, we rely on the security of the underlying cryptographic primitives. In more detail, the security of the key agreement scheme **KA** ensures that entities in $\mathcal{C}$ (who have access to the public keys of all clients, the private keys of the corrupted ones, and the transcript of the setup phase) cannot distinguish the actual pairwise keys from random values.

Additionally, the security of the **TJL** scheme ensures that entities in $\mathcal{C}$ cannot distinguish protected inputs $Y_{u,\tau}$ from random values. It also ensures that if entities in $\mathcal{C}$ have access to no more than $t-1$ shares of the client secret key sk_u (i.e. $|\mathcal{U}_{corr}| < t$), then entities in $\mathcal{C}$ cannot distinguish the shares held by the honest clients from random values.

Finally, the security of the secret sharing scheme **SS** ensures that, if entities in $\mathcal{C}$ have access to no more than $t-1$ shares of the masking seed $b_{u,\tau}$ (i.e. $|\mathcal{U}_{corr}| < t$), then they cannot distinguish the shares held by the honest clients from random values.

Therefore, the view of entities in $\mathcal{C}$ at the end of each FL round τ is computationally indistinguishable from a simulated view. Thus, the aggregator learns nothing more than the sum of the online clients' inputs if $|\mathcal{U}^{\tau}_{on}| \geq |\mathcal{U}^{\tau}_{shares}| \geq t$; Otherwise the aggregator learns nothing.

Security in the active model In the active model, the aggregator can additionally manipulate its inputs to the protocol. We focus on showing that the protocol is secure in the active model based on the definition of Aggregator Obliviousness. We do not address the problem of validating the results of the aggregation. We assume the existence of a Public-Key Infrastructure (PKI) to rule out attacks over the integrity of messages and Sybils.

The only messages the server distributes, other than the clients' public keys, are the encrypted shares which are forwarded from and to the clients. The server cannot modify the values of these encrypted shares thanks to the underlying authenticated encryption **AE** scheme. Therefore, the server's power in the protocol is limited to refraining from forwarding some of the shares. This will make clients reach some false conclusion about the set of online clients in the protocol. Note, importantly, that the server can give different views to different clients about which clients have failed. Therefore, the server can convince a set of honest clients to send the protected zero-value Y'_{τ} corresponding to a client u (i.e., $\mathcal{U} \setminus \mathcal{U}^{\tau}_{on} = \{u\}$) while asking another set of honest clients to send the shares of the blinding mask $b_{u,\tau}$. The server should not obtain the protected zero-value Y'_{τ} corresponding to a client u and its blinding mask $b_{u,\tau}$ at the same time. If that happened, the server can recover the masked input from Y'_{τ} and $Y_{u,\tau}$, then remove the mask using $b_{u,\tau}$.

Knowing that the server may collude with $n-t$ corrupted clients, it can obtain $n-t$ shares of Y'_{τ} s.t. $\mathcal{U} \setminus \mathcal{U}^{\tau}_{on} = \{u\}$ and $n-t$ shares of $b_{u,\tau}$. Additionally, the server can manipulate $\frac{t}{2}$ honest clients to think that user u has failed (i.e., to send shares of Y'_{τ}) and the other $\frac{t}{2}$ honest clients to think that u is online (i.e., to send shares of $b_{u,\tau}$). Hence, in total, the server can learn a maximum number of $n-t+\frac{t}{2}$ shares of Y'_{τ} and $b_{u,\tau}$ of the same user. Therefore, to ensure security, we require that $n-t+\frac{t}{2} < t \implies t > \frac{2n}{3}$. Hence the protocol can recover from up to $\frac{n}{3} - 1$ client failures.

Conclusion If we assume an honest-but-curious aggregator, it is sufficient to choose $t > \frac{n}{2}$ to guarantee security. Otherwise, if we assume that the aggregator actively manipulates the protocol messages, the threshold parameter should be set to $t > \frac{2n}{3}$.

10 Discussion

In this section, we discuss some of the main ideas to improve our protocol. First, we discuss a practical technique to reduce the computation time for the clients based on a **JL** scheme property. Second, we discuss how to add additional protection to our protocol to prevent other types of inference attacks.

Further performance optimisations at the client side The experimental study in Section 8 shows that clients spend around 45% of the total computation time in round *Encrypt* of the online phase (95% in case of no client failures) (see Table 4 in the Appendix). Most of the computation work in round *Encrypt* corresponds to the execution of the protection algorithm **TJL.Protect**. The most expensive operation in **TJL.Protect** consists of the computation of $H(\tau)^{sk_u} \mod N^2$ (see Equation 1). Since the computation of this term is independent from the client's input, this term can be pre-computed and once the client's input is known, the latter would only perform one modular multiplication. Indeed, authors of the **JL** scheme [17] call this property "on-the-fly" encryption. In our protocol, clients can benefit from the idle time when the server is performing the aggregation in round *Construct* to pre-compute this value. The improvement of such optimization can reduce the client's computation time by up to 40% (and by up to 90% in case of no client failures).

Protecting the aggregated model In federated learning, the aggregated machine learning model is usually considered as publicly available. Consequently, secure aggregation remains vulnerable against inference attacks on the aggregated model [33]. Although these attacks do not leak information originating from a specific client dataset (thanks to secure aggregation), such a problem remains important. Several solutions such as [18,37] investigate the use of differential privacy techniques [11] as an additional protection layer for secure aggregation in the context of federated learning. These techniques are complementary to our scheme and thus can also be integrated to our solution.

11 Related Works

Several solutions have been integrating secure aggregation (SA) with federated learning. These solutions rely on the use various cryptographic techniques including secure masking, multi-input functional encryption (MIFE), secret sharing, and additively-homomorphic encryption (AHE). Masking-based SA solutions such as [3, 4, 16, 18, 40] provide an efficient protection algorithm but incur high computation overhead since they usually require the execution of a new key setup process for each federated learning round. On the other hand, MIFE-based SA [39,41] enables the computation of weighted sums using the inner product function with lightweight operations. Similar to masking-based solutions, client keys should be generated for each FL round and therefore such solutions relies on an online trusted key dealer. Additionally, there exist SA solutions based on Shamir's secret sharing including [1, 10, 20, 26] that are fault-tolerant and decentralized by design. Neverttheless, these solutions incur a significant communication overhead. Last but not least, AHE-based secure aggregation solutions were initially used for smart-meter applications [15, 21, 32]. While such solutions allow for the use of a long-term key, they incur large communication overhead due to the large ciphertext size. To deal with this limitation, different solutions [22, 29, 42] propose to batch the client's model in federated learning. For instance, BatchCrypt [42] proposes an efficient encoding technique to quantize the machine learning parameters before encrypting them with the Paillier encryption scheme [28]. This encoding technique can be ported to our scheme to further reduce the communication overhead.

Failures of FL clients is a well known problem for secure aggregation schemes. Few solutions have been proposed to address this problem in the context of federated learning. For SA solutions based on MIFE, HybridAlpha [41] proposed to replace the weights of failed users by zeros. On the other hand, Bonawitz et al. [4] proposed the use of secure masking. Unfortunately, both solutions inherit the drawback of their underlying building block and thus require a complex key management process at runtime. We believe that our solution is the first secure and fault-tolerant aggregation scheme based on AHE and the use of long-term keys for federated learning applications.

12 Conclusions and Future Work

We have designed a secure and fault-tolerant aggregation protocol for federated learning. Firstly, we constructed a threshold-variant of Joye-Libert [17] secure aggregation scheme (**TJL**). The secure federated learning protocol uses the **TJL** scheme to protect FL clients' inputs and securely aggregate them even in the presence of up to $\frac{n}{3}$ failures. We show that our scheme outperforms the state-of-the-art fault-tolerant secure aggregation [4] in terms of computational cost by n orders of complexity (n being the number of total clients in the protocol).

As part of our future work, we aim to cover stronger threat models. Namely, we would like to ensure the correctness of the computation of the aggregate value when dealing with malicious users and/or aggregator.

References

[1] Constance Beguier and Eric W. Tramel. Safer: Sparse secure aggregation for federated learning, 2020.

[2] 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. Association for Computing Machinery, 2020.

[3] Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. *CoRR*, abs/1902.01046, 2019.

[4] 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. CCS '17, page 1175–1191, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Keith Bonawitz, Fariborz Salehi, Jakub Konečný, Brendan McMahan, and Marco Gruteser. Federated learning with autotuned communication-efficient secure aggregation. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, 2019.

[6] Léon Bottou. Stochastic learning. In Olivier Bousquet and Ulrike von Luxburg, editors, *Advanced Lectures on Machine Learning*, Lecture Notes in Artificial Intelligence, LNAI 3176, pages 146–168. Springer Verlag, Berlin, 2004.

[7] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9(6), Dec 2010.

[8] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 2006.

[9] Tassos Dimitriou and Mohamad Khattar Awad. Secure and scalable aggregation in the smart grid resilient against malicious entities. *Ad Hoc Networks*, 50, 2016.

[10] Ye Dong, Xiaojun Chen, Liyan Shen, and Dakui Wang. Eastfly: Efficient and secure ternary federated learning. *Computers & Security*, 94:101824, 2020.

[11] Cynthia Dwork. Differential privacy. In *Automa, Languages and Programming*. Springer Berlin Heidelberg, 2006.

[12] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[13] Morris J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, Gaithersburg, MD, USA, 2007.

[14] Ahmed Roushdy Elkordy and A. Salman Avestimehr. Secure aggregation with heterogeneous quantization in federated learning, 2020.

[15] Zekeriya Erkin and Gene Tsudik. Private computation of spatial and temporal power consumption with smart meters. In *Applied Cryptography and Network Security*, pages 561–577. Springer Berlin Heidelberg, 2012.

[16] Xiaojie Guo, Zheli Liu, Jin Li, Jiqiang Gao, Boyu Hou, Changyu Dong, and Thar Baker. Verifl: Communication-efficient and fast verifiable aggregation for federated learning. *IEEE Transactions on Information Forensics and Security*, 16, 2021.

[17] Marc Joye and Benoît Libert. A scalable scheme for privacy-preserving aggregation of time-series data. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2013.

[18] Peter Kairouz, Ziyu Liu, and Thomas Steinke. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*. PMLR, 2021.

[19] Ferhat Karakoç, Melek Önen, and Zeki Bilgin. Secure aggregation against malicious users. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*, SACMAT '21. Association for Computing Machinery, 2021.

[20] Youssef Khazbak, Tianxiang Tan, and Guohong Cao. Mlguard: Mitigating poisoning attacks in privacy preserving distributed collaborative learning. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020.

[21] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Privacy Enhancing Technologies*. Springer Berlin Heidelberg, 2011.

[22] Changchang Liu, Supriyo Chakraborty, and Dinesh Verma. *Secure Model Fusion for Distributed Learning Using Partial Homomorphic Encryption*. Springer International Publishing, 2019.

[23] E. Meijering. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 2002.

[24] 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.

[25] 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. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[26] 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. *CoRR*, abs/2101.02281, 2021.

[27] Takashi Nishide and Kouichi Sakurai. Distributed paillier cryptosystem without trusted dealer. In Yongwha Chung and Moti Yung, editors, *Information Security Applications*, pages 44–60, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[28] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — EUROCRYPT '99*, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[29] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13, 2018.

[30] Tal Rabin. A simplified approach to threshold and proactive rsa. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, Berlin, Heidelberg, 1998. Springer-Verlag.

[31] Adi Shamir. How to share a secret. *Commun. ACM*, 1979.

[32] Elaine Shi, T.-H Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. volume 2, 01 2011.

[33] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[34] Jinhyun So, Ramy E. Ali, Basak Guler, Jiantao Jiao, and Salman Avestimehr. Securing secure aggregation: Mitigating multi-round privacy leakage in federated learning. *CoRR*, abs/2106.03328, 2021.

[35] Jinhyun So, Başak Göler, and A. Salman Avestimehr. Byzantine-resilient secure federated learning. *IEEE Journal on Selected Areas in Communications*, 39, 2021.

[36] Jinhyun So, Başak Güler, and A. Salman Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. *IEEE Journal on Selected Areas in Information Theory*, 2, 2021.

[37] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19. Association for Computing Machinery, 2019.

[38] Thijs Veugen, Thomas Attema, and Gabriele Spini. An implementation of the paillier crypto system with threshold decryption without a trusted dealer. Cryptology ePrint Archive, Report 2019/1136, 2019. https://ia.cr/2019/1136.

[39] Danye Wu, Miao Pan, Zhiwei Xu, Yujun Zhang, and Zhu Han. Towards efficient secure aggregation for model update in federated learning. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020.

[40] Guowen Xu, Hongwei Li, Sen Liu, Kan Yang, and Xiaodong Lin. Verifynet: Secure and verifiable federated learning. *IEEE Transactions on Information Forensics and Security*, 15, 2020.

[41] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19. Association for Computing Machinery, 2019.

[42] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020.

A Detailed Measurements of the Running Time

	# Clients	Failures	Register	KeySetup	Encrypt	Construct
Client	100	0%	1.08 ms	1607 ms	976 ms	18.6 ms
		30%	0.86 ms	1583 ms	966 ms	1071 ms
	300	0%	0.90 ms	4730 ms	1234 ms	56.8 ms
		30%	0.94 ms	4718 ms	1233 ms	1642 ms
	600	0%	0.89 ms	9202 ms	1733 ms	109 ms
		30%	0.92 ms	8754 ms	1630 ms	2346 ms
Server	100	0%	0.005 ms	1.98 ms	1.97 ms	1235 ms
		10%	0.007 ms	1.75 ms	1.7 ms	25196 ms
		30%	0.009 ms	1.63 ms	1.65 ms	19176 ms
	300	0%	0.018 ms	16.6 ms	16.2 ms	7887 ms
		10%	0.009 ms	16.9 ms	15.9 ms	294910 ms
		30%	0.009 ms	17.4 ms	16.1 ms	226290 ms
	600	0%	0.016 ms	109 ms	102 ms	41057 ms
		10%	0.017 ms	116 ms	105 ms	1357778 ms
		30%	0.014 ms	96.7 ms	100 ms	962070 ms

Table 4: Wall-clock running time for the clients and the server for our protocol. The dimension is fixed to $m = 10000$.

B Detailed Measurements of the Data Transfer

# Clients	Failures		Register	KeySetup	Encrypt	Construct
100	0%	*sent*	0.13 KB	32.84 KB	62.47 KB	1.96 KB
		rcvd	– KB	45.73 KB	– KB	5.46 KB
		total	0.13 KB	78.57 KB	62.47 KB	7.42 KB
	30%	*sent*	0.13 KB	32.84 KB	62.46 KB	58.81 KB
		rcvd	– KB	45.73 KB	– KB	3.81 KB
		total	0.13 KB	78.57 KB	62.46 KB	62.62 KB
300	0%	*sent*	0.13 KB	135.95 KB	79.00 KB	5.86 KB
		rcvd	– KB	174.62 KB	– KB	16.50 KB
		total	0.13 KB	310.57 KB	79.00 KB	22.36 KB
	30%	*sent*	0.13 KB	135.95 KB	79.00 KB	67.09 KB
		rcvd	– KB	174.62 KB	– KB	11.53 KB
		total	0.13 KB	310.57 KB	79.00 KB	78.62 KB
600	0%	*sent*	0.13 KB	400.79 KB	97.30 KB	11.72 KB
		rcvd	– KB	478.13 KB	– KB	33.05 KB
		total	0.13 KB	878.92 KB	97.30 KB	44.77 KB
	30%	*sent*	0.13 KB	400.78 KB	97.30 KB	72.96 KB
		rcvd	– KB	478.13 KB	– KB	23.12 KB
		total	0.13 KB	878.91 KB	97.30 KB	96.08 KB

Table 5: Data transfer per client for our protocol. The dimension is fixed to $m = 10000$.