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

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

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

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

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 works propose to rely on a secure aggregation scheme that allows parties to encrypt their model updates before sending them to the central server that aggregate the encrypted inputs. 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. Using this new building block together with a 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 a ML model of 100, 000 parameters, trained with 600 clients, our protocol is 5.5x faster (1.6x faster in case of 180 clients drop) at the client and 1.3x faster at the server.

CCS CONCEPTS

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

KEYWORDS

Federated Learning, Secure Aggregation, Fault-Tolerance

ACM Reference Format:
Mohamad Mansouri, Melek Önen, and Wafa Ben Jaballah. 2018. Learning from Failures: Secure and Fault-Tolerant Aggregation for Federated Learning. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

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 server. The server computes the sum of clients' updates and sends the average back to the clients. The clients repeat the training process in several rounds so that a converged average model is reached. While federated learning is promising, as shown in [25, 26], a naïve use of such a scheme can still result in some leakage based on the exchange of model parameters and thus model updates also need to be protected.

Secure aggregation [27, 20] which ensures the aggregation of multiple parties' inputs without disclosing them individually, becomes a common solution to address this problem in federated learning. An aggregator receiving protected model updates from clients is still able to compute their aggregate. Unfortunately, the majority of secure aggregation solutions require all clients to be online [33, 18, 1, 10]. If a client fails to provide its protected input, the server will not be able to compute the sum.

In some federated learning applications, clients are mobile devices that may frequently encounter failures (for example, due to network connectivity problems) and hence existing secure aggregation solutions fall short to address such a problem. A previous work from Bonawitz et al. [5] 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 [10]: 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 [9] protocol executed by each pair of clients. Additionally, DH keys are shared among n clients using Shamir's secret sharing [32] 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 [6, 15, 4, 3, 37, 19, 35, 41, 17, 36].

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. Encryption schemes with long-term keys seem more promising for federated learning. Indeed, long-term keys are distributed once in the setup phase and then used to protect the clients' models in all the consecutive federated

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

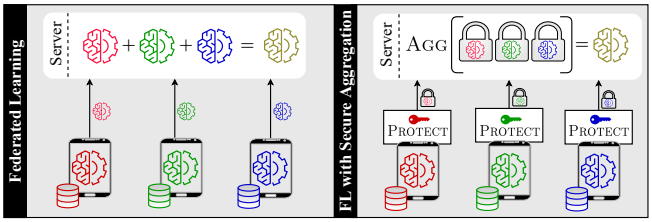


Figure 1: Illustration of federated learning with and without secure aggregation. The clients locally train the model on their own private dataset. Clients send their trained models (protected when using secure aggregation) to the server. The server aggregates the models and server learns only the aggregate when using secure aggregation.

learning rounds. This eliminates the need for a key agreement at each federated learning round.

Hence, we propose to use the Joye-Libert (JL) secure aggregation scheme [18] for federated learning since, it allows the protection of a user input with its own unique long-term key. On the other hand, unfortunately, similar to existing schemes, the JL scheme does not support client failures. Therefore, we propose to revisit the Joye-Libert (JL) scheme in order to cope with client failures and further make use of it to develop a secure and fault-tolerant aggregation scheme dedicated to federated learning applications. The new version of the scheme allows sharing the client's individual keys using Shamir's secret sharing [32] 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 clients. Compared to [5], 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 threshold version of the Joye-Libert scheme [18] by allowing clients to secret share their individual keys. The new scheme allows a set of available clients to compute the encryption of a zero-value, in case of someone drops, on behalf of the missing client.
- We propose a fault-tolerant secure aggregation protocol using our threshold Joye-Libert scheme. Our scheme completes the aggregation in two communication rounds (among the server and the clients) which is less than in [5] (four communication rounds).
- We implement a prototype of our protocol and evaluate it through a comparative study with the protocol in [5].
- Through our experimental study we show that our solution outperforms the solution in [5] in terms of computation at the client side and supports more failures with the same computational 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 passive and active adversary models.

2 BACKGROUND AND THREAT MODEL

Federated Learning. In federated learning (FL), a set $\mathcal{U}$ of FL clients locally train a common 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) [7]) 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 model parameters 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 $\mathcal{M}_T$ is learned after T rounds.

Although FL clients keep their own datasets private, studies have shown that adversaries who have access to the client's updated model $\mathcal{M}_{u,\tau+1}$ can infer information about its private dataset $\mathcal{D}_u$ [25, 26]. Hence the local models should remain confidential even against the FL server.

Secure Aggregation. Secure aggregation (SA) is a protocol that involves multiple users and one aggregator. Each user holds a private input and the aggregator computes the sum. It requires that the aggregator cannot learn more than the sum of the users' inputs. Secure aggregation is used for federated learning to privately aggregate the clients' updates in each federated learning round using the FL server as the aggregator. Figure 1 demonstrates a FL protocol with and without secure aggregation.

Threat model. We consider a threat model where an untrusted FL server colludes with some clients. Additionally, we consider some of the honest clients to unintentionally fail (i.e., drop from the protocol) in some federated learning rounds. The failures can happen at any stage of the protocol. The adversary (controlling the server and the colluding clients) is interested in discovering any private information about the individual inputs of the honest clients. We consider two adversary settings in which we later analyze the security of our protocol. (i) Passive model: the adversary correctly follows the protocol steps but tries to discover private information about clients' local models by looking at the protocol transcript. (ii) Active model: the adversary manipulates the messages in order to learn the clients' private information. We assume a trusted party to generate the public parameters of our protocol.

Attacks that aim to change the result of the aggregated data or to perform some denial of service are out of the scope. Additionally, we do not consider attacks where the adversary impersonates existing clients as they can be prevented by deploying a public key infrastructure with signed certificates. Note that this threat model is common among secure aggregation protocols and it is the same as the one adopted by [5] except for the dependency on a trusted party for the setup. We show later how to avoid this dependency.

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**) [32] 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 $u \in \mathcal{U}$. The user identifier 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 formula [24] 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 [31] and we denote it by **ISS**. The scheme shares a secret integer s in an 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 formula 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 generates key pairs from public parameter where $(c_u^{PK}, c_u^{SK}) = (g^a, a)$ and $a \xleftarrow{R} {}^\dagger\mathbb{Z}_q^*$.
- $c_{u,v} \leftarrow$**KA.Agree**$(pp, c_u^{SK}, c_v^{PK}, H)$: This algorithm uses the private key of user u, the public key of user v, and a hash function to generate a secret authentication and encryption key as $c_{u,v} = H((c_v^{PK})^{c_u^{SK}})$.

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 and to verify its integrity.

3.6 Joye-Libert Secure Aggregation Scheme

This secure aggregation scheme that we denote by **JL** is proposed in [18]. The scheme involves a trusted key-dealer, n users and the aggregator. **JL** can be defined as follows:

- $(sk_0, \{sk_u\}_{u \in \{1,..,n\}}, N, H) \leftarrow$ **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 the private input $x_{u,\tau} \in \mathbb{Z}_N$ for time period τ using secret key $sk_u \in \mathbb{Z}_{N^2}$. It outputs cipher $y_{u,\tau}$ such that:

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

- $X_\tau \leftarrow$ **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:

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

Theorem 3.1. *The scheme provides Aggregator Obliviousness security under the Decision Composite Residuosity (DCR) assumption [29] in the random oracle model and under the assumption that each user encrypts only one value per time period.*

Proof. Please refer to the original paper [18]. □

4 OUR APPROACH

The Joye-Libert (**JL**) scheme [18] allows n users to each encrypt its private input with a unique long-term pre-distributed key. The scheme is homomorphic on both, the messages and the encryption keys. This allows an aggregator holding the sum of the user keys to decrypt the aggregate of the messages. Since this secure aggregation scheme supports long-term keys, it features a significant advantage

[†] $\xleftarrow{R}$ means chosen uiformally at random

in terms of computation and communication as it allows using the same keys for all federated learning rounds. However, using **JL** scheme for federated learning introduces the following challenges:

- Client failures: It is common in federated learning that some clients drop in several federated learning rounds. When one or more clients do not provide a ciphertext for the federated learning round τ, the aggregation fails. This is because the aggregation key used to aggregate the ciphertext is equal to the sum of the user keys. If a client fails, its encryption key will not be involved in the aggregation. We deal with this problem by designing a threshold version of **JL** scheme to recover from dropouts.
- Larger inputs: In federated learning, the inputs are the parameters of the client's model and are of vector type. **JL** is originally designed to encrypt single integers. We extend **JL** to support vector inputs.
- No trusted key dealer: **JL** requires a key dealer to distribute some keys to the users and the aggregator. However, a trusted key dealer may not be feasible in federated learning applications. Therefore, we propose a decentralied key setup phase to distribute the keys.

*Threshold variant of **JL** scheme.* We design a threshold **JL** scheme, whereby clients can secretly share their individual keys with other clients so that when one or more clients fail, t out of n clients that are still online help provide a protected zero-value that is computed with the failed clients' individual keys. By computing the protected zero-value of the missing clients, the server can correctly compute the aggregated result.

***JL** scheme with vector inputs.* We propose to encode the vector elements in a single integer. Then, **JL** protection and aggregation algorithms are computed on the encoded integers. The vector sum is decoded after aggregation.

***JL** scheme with decentralized key setup.* To avoid relying on a trusted key dealer for generating the keys, we use a distributed key generation mechanism. We mainly use secure multi-party computation such that the n users and the aggregator each holds a random share of zero. Each user will use its share as a secret key so that the sum of the keys with the aggregator key equals to zero.

5 THRESHOLD JOYE-LIBERT 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 [8] into this context. This extended solution will mainly help the server recover failed users' inputs (which consists of the protection of the zero-value under each failed user's individual key) and hence compute the final aggregate value. 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 the user key is integer 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**, consists 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 $[-2^l, 2^l]$ and l is the number of bits of the modulus N. Indeed, similar to [28], 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 intended for different user $v \in \mathcal{U}$.
- $[y'_\tau]_u \leftarrow$ **TJL.SharedProtect**$(pp, \{[\Delta sk_v]_u\}_{v\in\mathcal{U}''}, \tau)$: 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 that their inputs should 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 protected 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 [28], 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

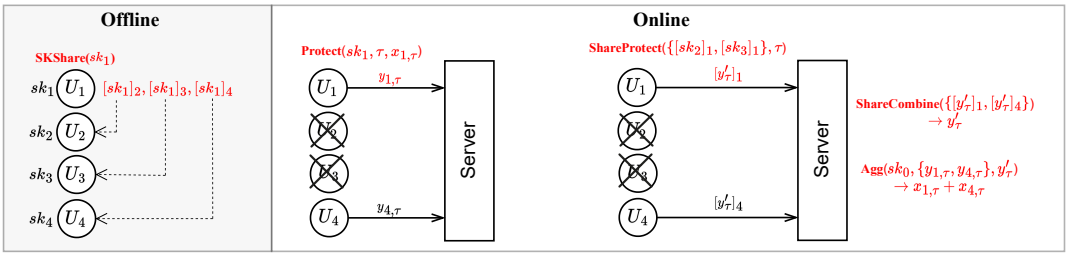


Figure 2: Demonstration of an execution of TJL scheme with four users ($n = 4$) and a reconstruction threshold $t = 2$.

$\mathcal{U}'' = \mathcal{U} \setminus \mathcal{U}'$ is the set of failed users. It computes:

$$\begin{aligned} y_\tau &= (\prod_{\forall u \in \mathcal{U}'} y_{u,\tau})^{\Delta^2} \cdot y'_\tau \mod N^2 \\ &= (1 + \Delta^2 \sum_{\forall u \in \mathcal{U}'} x_{u,\tau} N) H(\tau)^{\Delta^2 \sum_{\forall u \in \mathcal{U}'} sk_u} \cdot H(\tau)^{\Delta^2 \sum_{\forall u \in \mathcal{U}''} sk_u} \\ &= (1 + \Delta^2 \sum_{\forall u \in \mathcal{U}'} x_{u,\tau} N) H(\tau)^{\Delta^2 \sum_{\forall u \in \mathcal{U}} sk_u} \\ &= (1 + \Delta^2 \sum_{\forall u \in \mathcal{U}'} x_{u,\tau} N) H(\tau)^{-\Delta^2 sk_0} \end{aligned} \tag{3}$$

To decrypt the final result, the algorithm proceeds as follows:

$$V_\tau = H(\tau)^{\Delta^2 sk_0} \cdot y_\tau \qquad X_\tau = \frac{V_\tau - 1}{N\Delta^2} \mod N \tag{4}$$

Theorem 5.1. *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 (Theorem 3.1) and the security of the secret sharing scheme over integers which is also proved to be statistically secure in Theorem 1 in [39]. □

6 FAULT-TOLERANT SECURE AGGREGATION USING TJL

In this section, we describe the newly designed secure and fault-tolerant aggregation protocol based on the proposed **TJL** scheme. The protocol consists of a setup phase and an online phase each defined with two communication rounds. The setup phase is performed a single time, while the online phase is repeated for each federated learning round. We describe the details of each protocol phase.

6.1 The Setup Phase

The setup phase achieves two main goals: the registration of the clients and the distribution of the security keys. According to **TJL** scheme, a trusted key dealer is needed to generate the keys. However, it is not practical for some FL applications that the clients should request keys from a trusted party. To avoid the dependence on a key dealer, we propose a distributed method to setup the **TJL** user keys.

*Distribution of **TJL** keys.* We recall that the aggregator's key sk_0 allows the aggregator to recover the sum of the inputs from the set of users' ciphertexts (see Equation 3). The goal of this key is to protect the final aggregate. In the case of FL applications, the aggregated ML model is considered public so there is no need to hide it from adversaries. Therefore, we can set the aggregator's key to a public known value (for example zero). Indeed, the security proof in [18] considered 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. To generate the user keys, each two clients u and v agree using the **KA** scheme on a shared mutual key $sk_{u,v}$. Then client u computes its protection key $sk_u \leftarrow \sum_{v \in \mathcal{U}} (\delta_{u,v} \cdot sk_{u,v})$ where $\delta_{u,v} = 1$ when $u > v$, and $\delta_{u,v} = -1$ when $u < v$. The correctness of the protocol is preserved since:

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

Note that this distributed method allows the distribution of the **JL** 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 [28, 39]. In this work, we assume a trusted offline third party to distribute the public parameters and we consider the full independency on a trusted party as future work.

The protocol steps. During *Registration* step, clients register by sending their public keys and the aggregator broadcasts them to the other clients. Notice that each client generates two public keys, one used to create secret communication channels between clients and the other used to compute the **TJL** secret key. Later in the *Key Setup* step, each client uses the key agreement and the clients' public keys to agree on mutual channel keys $c_{u,v}$ and to compute its **TJL** key sk_u. It also creates secret shares of the **TJL** keys using **TJL.SKShare** and sends them to the corresponding clients (passing by the server through the authenticated channels). The specifications of the setup phase are shown in Figure 3.

6.2 The Online Phase

The online phase is composed of two communication rounds. In the first communication round, (i.e., *Encryption* step), the clients protect their inputs and sends them to the aggregator. In the second communication round (i.e, *Aggregation* step), the clients and the aggregator construct the ciphertext of the failed clients.

Secure Aggregation Protocol - Setup Phase

- **Setup - Registration:**

  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 $(c_u^{PK} \mid\mid s_u^{PK})$ 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 $sk_u \leftarrow \sum_{v \in \mathcal{U}} \delta_{u,v} \cdot sk_{u,v}$ where $\delta_{u,v} = 1$ when $u > v$, and $\delta_{u,v} = -1$ when $u < v$.
  - 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: $\epsilon_{u,v} \leftarrow$ **AE.enc**$(c_{u,v}, u \mid\mid v \mid\mid [sk_u]_v)$.
  - If any of the above operations fails, abort.
  - Send all the encrypted shares $\{\epsilon_{u,v}\}_{\forall v \in \mathcal{U}}$ 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 $\{(u, v, \epsilon_{u,v})\}_{\forall v \in \mathcal{U}}$.
  - Forward to each user $v \in \mathcal{U}$ its corresponding encrypted shares: $\{(u, v, \epsilon_{u,v})\}_{\forall u \in \mathcal{U}}$ and move to the online phase.

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

Figure 3: Detailed description of the setup phase of our secure and fault-tolerant aggregation protocol

Blidning inputs to ensure privacy. One problem with the **TJL** scheme is that its direct use does not guarantee privacy. The problem stems from the fact that the original scheme **JL** is privacy-preserving assuming that the user only provides a single ciphertext per time period (FL round τ) (see Section 3.6). However, let's assume the case where a client sends its protected input with some delay. The delay may cause the aggregator to request the online clients to construct the protected zero-value of the assumed failed client. In this case, two ciphertexts for the same time period are collected breaking the security assumption of the **JL** scheme. To deal with this problem, the client masks its input before encrypting it. The goal of the mask is to protect any leakage in case two ciphertexts of the same period are obtained. To remove these masks from the aggregated value, each client secretly shares its mask with all other clients. If the client survives the federated learning round, the online clients help construct its blinding mask. Otherwise, the clients construct the ciphertext of the zero-value using the **TJL** scheme.

Input vector encoding. The **TJL** scheme is originally designed to work with integers. In federated learning applications, FL clients send a vector of parameters instead of a single one. We therefore propose a dedicated encoding solution to encode a vector into a long 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 the elements of the vector are packed to form a large integer ω. The number of elements that ω can represent corresponds to $ptsize$ which denotes the plaintext size of the **TJL** scheme divided by the actual size of the extended element (i.e $\lfloor \frac{ptsize}{\log_2(R)+\log_2(n)} \rfloor$,$R$ being the maximum possible value for vector elements). Note that for **TJL** scheme, the plaintext size is equal to the half of the size of the user key ($ptsize = \frac{|sk_u|}{2}$).

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

To evaluate **TJL.Protect** and **TJL.ShareProtect** algorithms on vectors, the user input is first encoded. Then, the algorithm is applied on encoded integer. In the case where the user input vectors are too large to be encoded in a single integer of $ptsize$ size, the vectors are split into multiple vectors each of length $\lfloor \frac{ptsize}{\log_2(R)+\log_2(n)} \rfloor$ and encoded separately. The algorithms are then applied on each encoded vector. We use a counter c to generate a unique time period for each encoded part.

The protocol steps. In the *Encryption* step, the client generates a random seed b_u. Then, it extends it using a PRG generating the blinding mask B_u. The client first blinds its input with the mask then protects its with **TJL.Protect**. After that, the client secretly shares the seed b_u with other clients and sends its masked and protected input to the server.

In the *Aggregation* step, the client learns the list of failed clients. So, it computes **TJL.ShareProtect** using the sum of their TJL keys' shares. Then, it sends to the server the blinding mask share of each online client and the share of the protected zero-value corresponding to all the failed clients. The server constructs the blinding masks and the protected zero-value using **TJL.ShareCombine**. Finally, it aggregates using **TJL.Agg** to obtain the blinded sum which is unblinded by removing the clients' masks. The detailed specification of this phase is provided in Figure 4.

Secure Aggregation Protocol - Online Phase

- **Online - Encryption (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 - Aggregation (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 4: Detailed description of the online phase of our secure and fault-tolerant aggregation protocol

7 SECURITY ANALYSIS

In this section, we evaluate the security of our secure and fault-tolerant aggregation protocol and prove that it ensures privacy of the individual inputs in both the passive and active adversary setting (see Section 2) given a dedicated threshold t of honest clients.

Security in the passive 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 $C = \mathcal{U}_{corr} \cup \mathcal{S}$ ($\mathcal{S}$ represents the server). The view of C is computationally indistinguishable from a simulated view if the number of corrupted clients is less than the threshold t ($|\mathcal{U}_{corr}| = n - t < t$). 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 details, the security of the key agreement scheme **KA** ensures that entities in 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 of the honest clients from random values.

Additionally, the security of the **TJL** scheme ensures that entities in C cannot distinguish protected inputs $Y_{u,\tau}$ from random values. It also ensures that if entities in 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 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 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 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 $|\mathcal{U}_{on}^{\tau}| \geq |\mathcal{U}_{shares}^{\tau}| \geq t$; Otherwise the aggregator learns nothing.

Security in the active model. In the active model, the aggregator can additionally manipulate its inputs to the protocol. 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., $\mathcal{U} \setminus \mathcal{U}_{on}^{\tau} = \{u\}$) 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 $b_{u,\tau}$ for the same FL round τ. 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. $\mathcal{U} \setminus \mathcal{U}_{on}^{\tau} = \{u\}$ and $n-t$ shares of $b_{u,\tau}$. For the remaining t honest clients, the server can manipulate $\frac{t}{2}$ clients to think that client 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 client. 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 a passive adversary, 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 $t > \frac{2n}{3}$.

8 SCALABILITY COMPARISON WITH MASKING-BASED FL

In this section, we conduct a comparative analysis with the solution in [5] (see Table 1). We first describe the methodology used in [5] and the main technical differences. Then we analyze the scalability of our solution in terms of computation, communication, and storage at both the client and the server and compare it with [5]. We perform the complexity analysis with respect to the number of clients n and the dimension of the client's input m. We only present the analysis of the online phase of both protocols.

8.1 Masking-based Secure Aggregation

Bonawitz et al. Fault-Tolerant Solution [5]. A previous solution from Bonawitz et al. [5] proposes a fault-tolerant version of secure aggregation. Authors build their protocol over a masking scheme. In masking, client protect their inputs using one time-pad encryption (i.e. modular addition with a random mask). Bonawitz et al. defines client's masks such that their sum is equal to zero. The aggregator can obtain the sum by simply adding the protected inputs. To generate the mask M_u, each pair of clients (u, v) agree on a shared key $s_{u,v}$ using a key agreement protocol. The client mask M_u is then 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. To recover from failed clients, the protocol integrates a t-out-of-n secret sharing [32]. In more detail, each FL client sends shares of its key agreement secret keys to all other clients. This way, if client u fails, a set of t or more clients can help the FL server reconstruct all the key agreement secret keys of the missing client and thus obtain $s_{u,v}\ \forall\ v \in \mathcal{U}_{on}$. Then, the server can recompute the mask M_u of the missing client and add it to the aggregate (to ensure that the masks cancel out). Additionally, the solution adds another masking layer using a blinding mask $B_u = \mathbf{PRG}(b_u)$ generated from a randomly chosen seed b_u. 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 server computes the sum of the masked inputs received from the online clients. Then, it reconstructs b_u for all the online clients ($\forall u \in \mathcal{U}_{on}$) and it reconstruct the key agreement secret keys for all failed ones ($\forall v \in \mathcal{U} \setminus \mathcal{U}_{on}$). The aggregate is revealed by first adding M_u of every failed client then removing the blinding masks B_u of the online ones.

Table 1: Complexity analysis of the online phase of both protocols (our protocol vs [5]). n is the number of clients and m is the dimension of the client's input.

		Ours	CCS17 [5]
Computation	*Client*	$O(n^2+m)$	$O(n^2+nm)$
	Server	$O(n^2+nm)$	$O(n^2m)$
Communication	*Client*	$O(n+m)$	$O(n+m)$
	Server	$O(n^2+nm)$	$O(n^2+nm)$
Storage	*Client*	$O(n+m)$	$O(n+m)$
	Server	$O(n^2+nm)$	$O(n^2+m)$

Differences with our protocol. The main difference with our proposed protocol is that we replace the masking scheme with a threshold encryption scheme (**TJL**). The use of **TJL** improves the performance for both the clients and the aggregator thanks to the two important advantages of our approach over the masking-based one:

- Our approach supports the use of the same encryption keys for all federated learning rounds. This was not the case in masking since the masks should be renewed for each round. The generation of new masks costs each client n key agreements and n calls to a PRG (extending the seeds to a vector of size m). These operations add a computation complexity of $O(nm)$ on each client in [5].
- When multiple clients fail, Bonawitz et al. solution requires the server to reconstruct the key agreement secret key of each failed client independently. After constructing the secret key, the n key agreements are simulated to reconstruct the mask of each of the failed clients. In contrast, our solution allows the recovery of the aggregate using only a single reconstruction operation. This is because the **TJL** supports reconstructing a single value (Y'_τ) that represents the protected zero-value on behalf of all failed clients. Consequently, the scalability of the protocol with respect to the number of failures is improved by an order of n on the server side.

8.2 Scalability at the client

Communication. $O(n+m)$. The communication cost consists of: In *Encryption* step, (1) sending $O(n)$ shares of $b_{u,\tau}$ and receiving $O(n)$ shares, and (2) sending the encrypted client input which is $O(m)$; In *Aggregation* step, (3) sending $O(n)$ shares of $b_{u,\tau}$, and (4) if at least one client failed, sending the share of the protected zero-value $Y'_{u,\tau}$ which is $O(m)$. In total, the complexity is $O(n+m)$ which is the **same** as in [5].

Computation. $O(n^2+m)$. The computation cost consists of: In *Encryption* step, (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 *Aggregation* step, (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 [5] ($O(n^2+nm)$).

Storage. $O(n+m)$. The client must store the keys and shares of each other client as well as the data vector which results in a storage overhead of $O(n+m)$ which is the **same** as the one in [5].

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 $m = 100K$.

# 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

8.3 Scalability at the server

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 [5].

Computation. $O(n^2 + nm)$. The server's computation cost is insignificant in *Encryption* step since the server only forwards messages. In *Aggregation* step it consists of: (1) computing the product of the received ciphertexts which corresponds to $O(nm)$. (2) reconstructing t out of n shares of $b_{u,\tau}$ for each online client u which is $O(n^2)$, (3) extending the seed $b_{u,\tau}$ to the dimension of the client's input (using **PRG**) which is $O(nm)$, if at least one client failed, (4) constructing the protected zero-value Y'_τ from its t shares which is $O(nm)$, and (5) aggregating the ciphertexts and unmasking the result which is $O(nm)$. The total computation cost equals to $O(n^2 + nm)$ which is **lower** than the one in [5] ($O(n^2 m)$).

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 clients 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 [5] ($O(n^2 + m)$) due to the larger size of the **TJL** shares with respect to shares of the key agreement keys.

9 EXPERIMENTAL EVALUATION

We have implemented a prototype of our protocol and a prototype of the protocol presented in [5] using Python programing language* and conducted an experimental evaluation. We describe the implementation details of the various cryptographic primitives in Appendix A. We further run several experiments with our protocol and the one in [5] while varying the number of the clients $n = \{100, 300, 600\}$. We also use different dimensions for the clients' inputs $m = \{1K, 10K, 100K\}$ and different client failure rates $f = \{0\%, 10\%, 30\%\}$. The measurements were performed using a single threaded process on a machine with an Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz processor and 32 GB of RAM.

*We plan to make these prototypes available for the public

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 $n = 600, m = 100K, f = 10\%$.

# 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%)

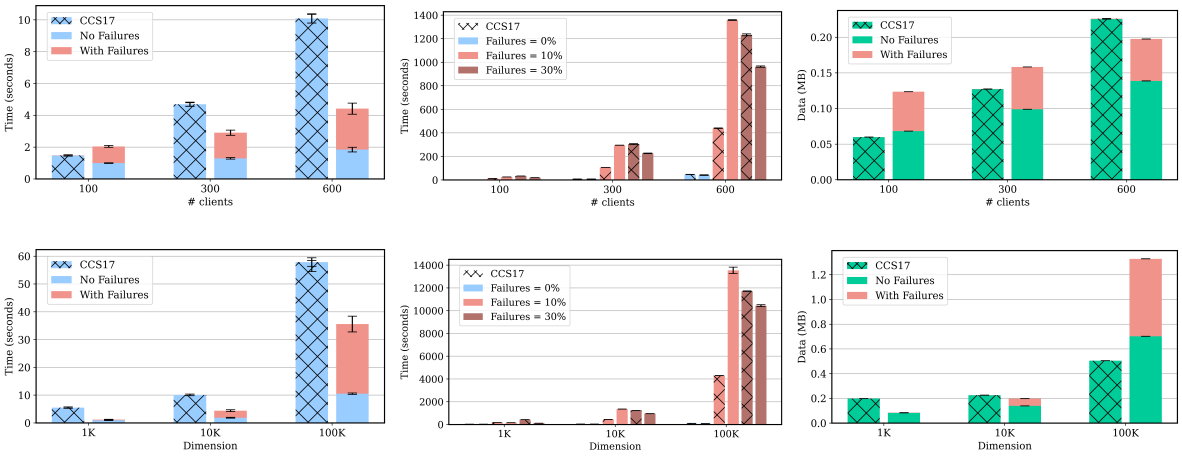
Running time for clients. We plot the wall-clock running time for the clients in both protocols (i.e., our protocol and the one in [5]) in Figure 5a. Our protocol shows better running time in most of the measured scenarios. Additionally, our protocol scales better with respect to the increasing number of clients (i.e., our solution is ×1.5 faster with 100 clients and ×5.5 faster with 600). This confirms the study in Section 8.2. Notice that our protocol has a constant overhead when client failures occur w.r.t. the ratio of client failures (see Table 2). This is expected since a client u computes a single value $[Y'_\tau]_u$ 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 [5]) in Figure 5b. 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 scalability in terms of ratio of failures. It is worth to mention that the reconstruction operation is heavier in our protocol since Lagrange formula is computed on the exponent. This is why the server overhead is lower in [5] in the case of few failures (10% of the clients failed). However, this overhead is constant in our protocol w.r.t. the number of failed clients (i.e., construction of single value Y'_τ that represents all failed clients). Therefore, it is more suitable in the case of many clients fail. We show the detailed running time (per protocol round) in Table 4 in Appendix B.

Data transfer. We plot the total data transfer (sent and received) per client in both protocols (i.e., our protocol and the protocol in [5]) in Figure 5c (the data transfer at the server is equal to that of a client multiplied by the number of client). When running with clients input of dimension $m = 10K$, 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 is mainly because of the larger size of the ciphertext (two times the size of the plaintext) with respect to masking. We stress that the data transfer remains acceptable as it is around 1.2MB for $m = 100K$. We show detailed measurements per protocol round in Table 5 in Appendix C.

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,



(a) Client Runtime **(b) Server Runtime** **(c) Bandwidth**

Figure 5: The wall-clock running time (a,b) and the total data transfer sent and received (c). The measurements are performed using a single-threaded python implementation of our solution and the solution in [5] (only the online phase time is shown). When varying the number of clients, we fix the input dimension to $m = 10$K and when varying the dimension we fix the number of clients to $n = 600$. Bars represent the average value based on 10 runs and the error margins represent the standard deviation.

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 9 shows that clients spend around 45% of the total computation time in *Encryption* step of the online phase (95% in case of no client failures) (see Table 4 in the Appendix). Most of the computation work in *Encryption* step 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 [18] 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 *Aggregation* step to pre-compute this value. The improvement of such optimization can reduce the end-to-end execution time of the protocol.

Further scalability improvements. We presented our protocol in a complete connected graph of clients (i.e. all clients send secret shares to all the other clients on the network). We believe that a fully connected graph is not necessary to guarantee correctness and security. A user can be simply be connected to k neighbors such that all users form a connected graph. Bell et al. [3] propose a method to build these graphs with $k = log(n)$. The authors apply their solution to Bonawitz et al. [5] protocol and achieved a better scalability where $O(n)$ operations are replaced by $O(k)$. The same approach can be applied to our protocol as it is independent from the encryption scheme being used.

Protecting the aggregated model. In federated learning, the aggregated machine learning model is public. Consequently, secure aggregation remains vulnerable against inference attacks on the aggregated model [34]. 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 [19, 38] investigate the use of differential privacy techniques [12] 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 [5, 4, 19, 41, 17] 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 [42, 40] 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 [2, 11, 21, 27] that are fault-tolerant and decentralized by design. Nevertheless, these solutions incur a significant communication overhead. Last but not least, AHE-based secure aggregation solutions were initially used for smart-meter applications [16, 33, 22]. 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 [23, 43, 30] propose to batch the client's model in federated learning. For instance, BatchCrypt [43] proposes an efficient encoding technique to quantize the machine learning parameters before encrypting them with the Paillier

encryption scheme [29]. 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 [42] proposed to replace the weights of failed users by zeros. On the other hand, Bonawitz et al. [5] 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 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 [18] 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 [5] 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] Gergely Ács and Claude Castelluccia. 2011. I have a dream! (differentially private smart metering). In *Information Hiding*. Springer Berlin Heidelberg.

[2] Constance Beguier and Eric W. Tramel. 2020. Safer: sparse secure aggregation for federated learning. (2020). arXiv: 2007.14861 [stat.ML].

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

[4] Kallista A. Bonawitz et al. 2019. Towards federated learning at scale: system design. *CoRR*, abs/1902.01046.

[5] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In (CCS '17). Association for Computing Machinery, New York, NY, USA, 1175–1191. ISBN: 9781450349468.

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

[7] Léon Bottou. 2004. Stochastic learning. In *Advanced Lectures on Machine Learning*. Lecture Notes in Artificial Intelligence, LNAI 3176. Olivier Bousquet and Ulrike von Luxburg, (Eds.) Springer Verlag, Berlin, 146–168. http://leon.bottou.org/papers/bottou-mlss-2004.

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

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

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

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

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

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

[14] Morris J. Dworkin. 2007. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep. Gaithersburg, MD, USA.

[15] Ahmed Roushdy Elkordy and A. Salman Avestimehr. 2020. Secure aggregation with heterogeneous quantization in federated learning. (2020).

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

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

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

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

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

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

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

[23] 2019. *Secure model fusion for distributed learning using partial homomorphic encryption. Policy-Based Autonomic Data Governance*. Springer International Publishing.

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

[25] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. 2019. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, 691–706. DOI: 10.1109/SP.2019.00029.

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

[27] Thien Duc Nguyen et al. 2021. FLGUARD: secure and private federated learning. *CoRR*, abs/2101.02281.

[28] Takashi Nishide and Kouichi Sakurai. 2011. Distributed paillier cryptosystem without trusted dealer. In *Information Security Applications*. Yongwha Chung and Moti Yung, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60. ISBN: 978-3-642-17955-6.

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

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

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

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

[33] Elaine Shi, T.-H Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-preserving aggregation of time-series data. In vol. 2. (Jan. 2011).

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

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

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

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

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

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

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

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

[42] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. 2019. 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.

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

A IMPLEMENTATION DETAILS

We use the same implementation and parameters for the building blocks of our protocol and the protocol proposed in [5].

- *Pseudo-Random Generator (**PRG**)*: We use AES encryption [13] 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 [14] 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.

B DETAILED MEASUREMENTS OF THE RUNNING TIME

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

	# Clients	Failures	Registeration	KeySetup	Encryption	Aggregation
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

C DETAILED MEASUREMENTS OF THE DATA TRANSFER

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

# Clients	Failures		Registration	KeySetup	Encryption	Aggregation
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