# Let Them Drop: Scalable and Efficient Federated Learning Solutions Agnostic to Stragglers

Riccardo Taiello
Inria / EURECOM / Université Côte d'Azur
Sophia Antipolis, France
riccardo.taiello@inria.fr

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

Clémentine Gritti
INSA Lyon / Inria
Villeurbanne, France
clementine.gritti@insa-lyon.fr

Marco Lorenzi
Inria / Université Côte d'Azur
Sophia Antipolis, France
marco.lorenzi@inria.fr

## ABSTRACT

Secure Aggregation (SA) stands as a crucial component in modern Federated Learning (FL) systems, facilitating collaborative training of a global machine learning model while protecting the privacy of individual clients' local datasets. Many existing SA protocols described in the FL literature operate synchronously, leading to notable runtime slowdowns due to the presence of *stragglers* (i.e. late-arriving clients). To address this challenge, one common approach is to consider stragglers as client failures and use SA solutions that are robust against dropouts. While this approach indeed seems to work, it unfortunately affects the performance of the protocol as its cost strongly depends on the dropout ratio and this ratio has increased significantly when taking stragglers into account. Another approach explored in the literature to address stragglers is to introduce asynchronicity into the FL system. Very few SA solutions exist in this setting and currently suffer from high overhead. In this paper, similar to related work, we propose to handle stragglers as client failures but design SA solutions that do not depend on the dropout ratio so that an unavoidable increase on this metric does not affect the performance of the solution. We first introduce **Eagle**, a synchronous SA scheme designed not to depend on the client failures but on the online users' inputs only. This approach offers better computation and communication costs compared to existing solutions under realistic settings where the number of stragglers is high. We then propose **Owl**, the first SA solution that is suitable for the asynchronous setting and once again considers online clients' contributions only. We implement both solutions and show that: *(i)* in a synchronous FL with realistic dropout rates (taking potential stragglers into account), **Eagle** outperforms the best SA solution, namely **Flamingo**, by ×4; *(ii)* In the asynchronous setting, **Owl** exhibits the best performance compared to the state-of-the-art solution **LightSecAgg** .

## CCS CONCEPTS

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

## KEYWORDS

Secure Aggregation, Synchronous and Asynchronous Federated Learning

**ACM Reference Format:**
Riccardo Taiello, Melek Önen, Clémentine Gritti, and Marco Lorenzi. 2024. Let Them Drop: Scalable and Efficient Federated Learning Solutions Agnostic to Stragglers. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30-August 2, 2024, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3664476.3664488

## 1 INTRODUCTION

Federated Learning (FL) [18] is a popular framework enabling multiple clients to collaborate in training a common machine learning model without sharing their local data. In centralized FL, the primary server initializes the parameters of a global model and sends them to the clients for optimization with respect to the local data. The locally trained parameters are transmitted to the server and aggregated (e.g. through weighted averaging) to produce a new global model for the next FL round.

Recent studies [19, 28] show that even sharing local model parameters may expose some information about the clients' training data, through various attacks such as membership inference or model inversion. A popular solution to tackle such attacks is Secure Aggregation (SA), which ensures that the global model's parameters are computed through the aggregation of the individual ones without disclosing them individually. Informally, each client first protects its local parameters and sends them to the server, and the server computes the aggregated parameters and shares them back to all the clients. The underlying privacy protection technique usually consists of either secure masking, additively homomorphic encryption, or differential privacy mechanisms [17].

As pointed out in [17], initial SA solutions were strongly relying on the online presence of all FL clients and even a single client failure, referred to as *client dropout*, was resulting in the complete failure of the aggregation protocol. To cope with this problem of robustness, many works [1, 3, 9, 15, 16, 29] propose to initially secret share clients' keying material with the others so that whenever


*ARES 2024, July 30-August 2, 2024, Vienna, Austria*

ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3664488

| | Client Comp. | Client Comm. | Online Rounds | FL Type |
|---|---|---|---|---|
| **SecAgg** [3] | $O(n^2 + nd)$ | $O(n + d)$ | 4 | SyncFL |
| **FTSA** [16] | $O(n^2 + n\log(n)d)$ | $O(n + d)$ | 3 | SyncFL |
| **Eagle** | $\boldsymbol{O(n\log(n) + d)}$ | $O(n + d)$ | 3 | SyncFL |
| **LightSecAgg** [30] | $O(n^2\frac{d}{(1-\delta)n-t} + d)$ | $O(n\frac{d}{(1-\delta)n-t} + d)$ | 3 | AsyncFL |
| **Owl** | $\boldsymbol{O(n^2 + d)}$ | $\boldsymbol{O(n + d)}$ | 3 | AsyncFL |

**Table 1: Complexity analysis for one round ($n$: number of clients; $\delta$: fraction of dropped clients; $t$: threshold value; $d$: input dimension).**

a client failure occurs, the remaining online clients can collaborate to reconstruct the dropped client's material and complete the aggregation operation correctly.

While these solutions have indeed been proven robust against client dropouts, their security is only valid when clients are synchronized and share their parameters on an FL-round basis. Unfortunately, a synchronous FL (SyncFL) setting encounters challenges in heterogeneous environments whereby slow, late-arriving clients, known as *stragglers* [7, 20, 32], can be detrimental to the overall system performance.

Very few solutions under SyncFL settings, namely [2, 33], address this challenge and employ a technique known as *over-selection*. In this approach, a larger pool of clients is initially engaged so that potential stragglers are inherently avoided. If this approach is adopted in the context of SA in SyncFL, then the dropout rate needs to be set as the sum of the potential ratio of stragglers and the actual client failures. In practical FL deployments, a dropout rate, including the ratio of stragglers, is expected to be around 30% [2, 20]. Unfortunately, this non-negligible ratio will result in a significant increase in SA parameters and consequently in a significant overhead both at the server [1, 3] and at the client [15, 16].

Asynchronous FL (AsyncFL) [7, 20, 32] modifies SyncFL by taking into account clients' model updates as soon as they arrive to the server. This allows to leverage the impact of stragglers, that do not block the system. Nevertheless, as previously mentioned in [30], existing SA solutions become insecure in such AsyncFL settings.

*Contributions.* In this paper, we cope with the problem of stragglers and propose two new SA protocols that address the aforementioned challenges inherent to realistic FL systems. More specifically:

- In the context of SyncFL, we significantly reduce the computation and communication overheads of FL clients and server through a new protocol named **Eagle**. More specifically, **Eagle** ignores dropped clients (including stragglers) and hence supports realistic dropout rates (from 10% to 30%). The performance improvement comes from a variant of the Threshold Joye-Libert scheme (**TJL**) proposed in [16].
- We develop a second protocol, named **Owl**, tailored to the AsyncFL setting. Similar to **Eagle**, **Owl** does not need to be aware of dropped clients and stragglers to complete the aggregation. We show that **Owl** is more efficient than the unique existing work [30]. Moreover, **Owl** is particularly suitable for deep learning models with large numbers of parameters.
- We conduct an extensive performance study and compare these two newly proposed schemes with relevant state-of-the-art solutions. Table 1 shows the asymptotic improvements of our two solutions compared to existing SA schemes. Especially, **Eagle** theoretically outperforms **SecAgg** and **FTSA**, whereas **Owl** asymptotically and experimentally shows better performance than **LightSecAgg** .

*Road Map.* The paper is organized as follows. Section 2 provides an overview of the so-called stragglers in synchronous and asynchronous FL settings, along with SA definition and threat model. Section 3 describes the required cryptographic primitives, including our new version of the **TJL** scheme. Section 4 and 5 describe our solutions, namely **Eagle** and **Owl**. Section 6 presents the works related to ours. Section 7 provides the complexity analysis of both our solutions and related works. Section 8 reports the experimental results of **Eagle** and **Owl**. Section 9 concludes our paper.

*Notations.* We provide the notations used throughout our paper in Table 2.

## 2 BACKGROUND

*Synchronous Federated Learning.* As introduced by McMahan et al. [18], FL consists of a distributed machine learning paradigm where a set $\mathcal{U}$ of clients ($|\mathcal{U}| = n_{tot}$) collaboratively trains a global model $\vec{x} \in \mathbb{R}^d$ under the guidance of a server. One of the first and popular methods used to train a FL model is FedAvg [18]. Within FedAvg, at each FL round $\tau$, the server defines a subset $\mathcal{U}^{(\tau)} \subseteq \mathcal{U}$ of clients ($|\mathcal{U}^{(\tau)}| = n \leq n_{tot}$) through *client selection* [6, 7, 18]. Each client $u \in \mathcal{U}^{(\tau)}$ trains the model $\vec{x}_{u,\tau}$ on its private local data $\mathcal{D}_u$, for example, through Stochastic Gradient Descent (SGD) [26]. Upon completion of the local training, the client forwards its updated model $\vec{x}_{u,\tau+1}$ to the server. When the server receives all the updated models from all clients in $\mathcal{U}^{(\tau)}$, it proceeds to the aggregation step by computing the average of these models: $\vec{x}_{\tau+1} \leftarrow \frac{1}{n}\sum_{u\in\mathcal{U}^{(\tau)}} \vec{x}_{u,\tau+1}$. This iterative process proceeds until the global model $\vec{x}$ reaches some desired level of accuracy. This approach to FL works under the Synchronous FL (SyncFL) setting whereby FL clients are synchronized and participate on a round-by-round basis. Usually, $n_{tot} \in [10^6, 10^{10}]$ and $n \in [50, 5000]$ [10].

*Asynchronous Federated Learning.* Asynchronous FL (AsyncFL) allows clients not to synchronise when training their local models. FedBuff [20] is introduced as a *buffered* asynchronous framework whereby the server stores local models received from clients in a

| Symbol | Description |
|---|---|
| $n_{tot}$ | total number of clients |
| $n$ | number of selected clients/ buffer size |
| $\delta$ | fraction of dropped clients |
| $t$ | number of honest online clients |
| $d$ | size of input |
| $\Delta$ | value set as equal to $n$! |
| $\mathcal{U}$ | set of clients |
| $\mathcal{U}^{(\tau)}$ | set of selected clients |
| $\mathcal{U}_{on}$ | set of online clients |
| $\mathcal{U}_{shares}$ | set of honest online clients |
| $x_u$ | client's (scalar) input |
| $\vec{x}_u$ | client's (vector) input |
| $y_u$ | protected client's (scalar) input |
| $\vec{y}_u$ | protected client's (vector) input |
| $y'$ | protected (scalar) zero value |
| $\vec{y}'$ | protected (vector) of zero values |
| $x$ | aggregate (scalar) |
| $\vec{x}$ | aggregate (vector) |
| $\tau$ | current FL round |
| $\tau_0$ | value set as equal to 0 |
| $\tau_u$ | current FL round of client $u$ |
| $[s]$ | share produced by secret sharing scheme |
| $N$ | modulus |
| $R$ | plaintext size |
| $\lambda$ | security parameter |
| $\xleftarrow{R}$ | chosen uniformly at random |

**Table 2: Notations.**

buffer and updates the global model whenever this buffer is full. Each client $u$ defines its local $\tau_u$ and each local update is denoted as $\vec{x}_{u,\tau_u}$. Note that the training round is specific to each client: namely, for another client $v \neq u$, $\tau_v \neq \tau_u$. To enhance readability, we denote $n$ as the buffer size in the context of AsyncFL.

*Stragglers.* In a SyncFL setting, all participating clients first update their local model, and then the server obtains a new global model by aggregation. This, in practice, means that all clients must synchronize with respect to the same FL round $\tau$. As a result, potentially wasted resources and important network delays might be caused by *stragglers* who require longer training times. *Over-selection* (typically by 30%) is a way to manage stragglers [2]. For example, if $n = 1000$ *selected* clients are needed to produce an accurate global model, $n' = 1300$ clients should be *over-selected.* The FL round $\tau$ ends when the fastest 1000 clients submit their local model updates while the slowest 300 clients are treated as *dropped.* In a buffered AsyncFL setting, the global model is simply updated as soon as the buffer is full with new local models, without specifically waiting for the stragglers [7, 32].

*Secure Aggregation.* Although FL clients keep their own datasets $\mathcal{D}_u$ private during training, adversaries who have access to the clients' updated model parameters can infer information about $\mathcal{D}_u$ [19, 28]. Hence, the local models should remain confidential even against the FL server. As already shown in [1, 3, 9, 16, 29], a potential solution for this problem is Secure Aggregation (SA). SA typically involves multiple clients and one *aggregator.* Each client possesses some private input and the aggregator calculates the sum of these inputs. The aggregator learns nothing more than the aggregated sum, thereby preserving the privacy of individual inputs. SA has easily found applications in FL since clients' local model parameters are protected and the only information the FL server (i.e. the aggregator) has access to is the global model parameters. Note that all FL protocols implementing SA only work with the assumption of SyncFL. So et al. [30] show that existing SA-based FL solutions do not work in an AsyncFL setting since clients' individual inputs can be leaked if there is no synchronisation.

*SA Threat Model and Security.* Similar to the related work [3, 15, 16, 30], we consider two potential adversaries:

- *The aggregator (i.e. the FL server)*: *(i)* in the *honest-but-curious* model, the aggregator does not modify any inputs to the protocol but still tries to learn private information about clients' local models, *(ii)* in the *active* model, the aggregator may manipulate the exchanged data in order to learn the clients' individual inputs.
- *The FL client*: There are $t$ honest clients and up to $n-t$ clients may collude with each other and/or with the aggregator. We assume that colluding clients only share their private information with each other and do not consider the case whereby they also try to manipulate the aggregated outcome.

# 3 BUILDING BLOCKS

## 3.1 Joye-Libert Secure Aggregation Scheme

The Joye-Libert scheme (**JL**) [8], involving a Trusted Dealer (TD), $n$ clients and one aggregator, is defined as follows:

- $(sk_0, \{sk_u\}_{u\in[1,n]}, N, H) \leftarrow$ **JL.Setup**$(\lambda)$: Given security parameter $\lambda$, this algorithm generates two large and equal-size prime numbers $p$ and $q$ and sets $N = pq$. It randomly generates $n$ secret keys $sk_u \xleftarrow{R} \mathbb{Z}_{N^2}$ and sets the aggregator key $sk_0 = -\sum_{u=1}^{n} sk_u$. 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 $\tau$ using secret key $sk_u \in \mathbb{Z}_{N^2}$. It outputs the cipher $y_{u,\tau} = (1 + x_{u,\tau}N) \cdot H(\tau)^{sk_u} \mod N^2$.
- $x_\tau \leftarrow$ **JL.Agg**$(pp, sk_0, \tau, \{y_{u,\tau}\}_{u\in[1,n]})$: This algorithm aggregates the $n$ ciphers received at time period $\tau$ to obtain $y_\tau = \prod_1^n y_{u,\tau}$ and decrypts the result $x_\tau = \sum_1^n x_{u,\tau} = \frac{H(\tau)^{sk_0} \cdot y_\tau - 1}{N} \mod N$.

The **JL** scheme ensures Aggregator Obliviousnes under the Decision Composite Residuosity (DCR) assumption [22] in the random oracle model and assuming that each client encrypts only one value per time period [8].

## 3.2 Threshold Joye-Libert SA Scheme

In this section, we elaborate on a new variant of the Threshold **JL** scheme (**TJL**) [16]. In the original **TJL**, clients assist the aggregator in recovering the inputs of failed clients, which consist of the protected zero value encrypted under the failed client's individual key. This process allows for the computation of the final aggregate value. The **TJL** proposed here is a slightly modified version that utilizes the same primitive but instead of reconstructing the encrypted zero value for dropped clients, it reconstructs the aggregated zero-value for online clients. This approach helps us remove the need for defining an aggregation key in advance. Instead, an on-the-fly, per-round aggregation key is built based on the actual online clients at the specific round.

**TJL** cannot directly use the standard Shamir Secret Sharing scheme (**SS**) [27] because $sk_u$ is defined in $\mathbb{Z}^*_{\phi(N^2)}$ and $\phi(N^2)$ is not known to the clients (see Section 3.1 [16]). Hence, the solution uses the Integer version of **SS** (**ISS**) [25], which is defined over integers rather than in a field (see Section 3.2 [16]). Informally, **ISS.Share** is run to split the secret key $sk_u$ into $n$ shares while **ISS.Recon** is called to recover the key given at least $t$ shares.

The **TJL** scheme consists of the following PPT algorithms:

• $(sk_0, \{sk_u\}_{u\in[1,n]}, N, H) \leftarrow$ **TJL.Setup**$(\lambda, \sigma)$: Given a security parameter $\lambda$, this algorithm essentially calls the original **JL.Setup**$(\lambda)$ and outputs the aggregator key, one secret key per client, and the public parameters. Additionally, it sets the security parameter of the **ISS** scheme to $\sigma$.

• $\{(v, [\Delta sk_u]_v)\}_{\forall v\in\mathcal{U}} \leftarrow$ **TJL.SKShare**$(sk_u, t, \mathcal{U})$: Upon input of client $u$'s secret key $sk_u$, this algorithm calls **ISS.Share** where the interval of the secret is $\mathbb{Z}_{N^2}$.

• $y_{u,\tau} \leftarrow$ **TJL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$: This algorithm primarily calls **JL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$ and outputs the ciphertext $y_{u,\tau}$.

• $[y'_\tau]_u \leftarrow$ **TJL.ShareProtect**$(pp, \{[\Delta sk_v]_u\}_{v\in\mathcal{U}_{on}}, \tau)$: This algorithm protects a zero-value using client $u$'s shares of all online clients secret keys (i.e. $v \in \mathcal{U}_{on}$). It calls **JL.Protect**$(pp, -\sum_{v\in\mathcal{U}_{on}}[\Delta sk_v]_u, \tau, 0)$ and outputs $[y'_\tau]_u = H(\tau)^{-\sum_{v\in\mathcal{U}_{on}}[\Delta sk_v]_u} \mod N^2$.

• $y'_\tau \leftarrow$ **TJL.ShareCombine**$(\{(u, [y'_\tau]_u, n)\}_{u\in\mathcal{U}_{shares}}, \mathcal{U}_{shares}, t)$: This algorithm combines $t$-out-of-$n$ protected shares of the protected zero-value for time period $\tau$ and clients in $\mathcal{U}_{shares} \subseteq \mathcal{U}_{on}$ such that $|\mathcal{U}_{shares}| \geq t$ and $\Delta = n!$. It executes the Lagrange interpolation on the exponent to get $y'_\tau = \prod_{u\in\mathcal{U}_{shares}} ([y'_\tau]_u)^{\mu_u} = H(\tau)^{-\Delta^2\sum_{v\in\mathcal{U}_{on}} sk_v}$ where the $\mu_u$ coefficients are defined in **ISS.Recon**.

• $x_\tau \leftarrow$ **TJL.Agg**$(pp, sk_0, \tau, \{y_{u,\tau}\}_{u\in\mathcal{U}_{on}}, y'_\tau)$: Given the public parameters $pp$, the aggregation key $sk_0$ (set to 0), the individual ciphertexts of online clients (i.e. $u \in \mathcal{U}_{on}$), and the ciphertexts of the zero-value corresponding to the clients $\mathcal{U}_{on}$, this algorithm aggregates the ciphertexts for time period $\tau$. It first multiplies the inputs for all clients in $\mathcal{U}_{on}$, raises them to the power of $\Delta^2$, and multiplies the result with the ciphertext of the zero-value to get $y_\tau = (\prod_{u\in\mathcal{U}_{on}} y_{u,\tau})^{\Delta^2} \cdot y'_\tau \cdot H(\tau)^{sk_0} \mod N^2$. To decrypt the final result, the algorithm calculates $x_\tau = \frac{y_\tau - 1}{N\Delta^2} \mod N$.

The **TJL** scheme provides AO under the DCR assumption in the random oracle model if the number of corrupted clients is strictly less than $t$ [16].

## 4 EAGLE IN SYNCFL

We present **Eagle**, a fault-tolerant SA solution in the context of a synchronous setting. The design principles are the following: (1) The first aim is to not depend on dropped clients/stragglers anymore and to consider online clients' inputs only. We hence eliminate the need for blinding masks as in existing SA solutions [1, 3] and significantly reduce the communication cost at the client side. (2) Instead of reconstructing the actual model parameters (which are usually assumed to be numerous), only one key is periodically

**Parties:** Server and selected clients in $\mathcal{U}^{(\tau)}$, such that $|\mathcal{U}^{(\tau)}| = n$
**Public Parameters:** Generate the public parameters $pp^{KA} \leftarrow$ **KA.Param**$(\lambda)$, $(\perp, \perp, N_0, H_0, \sigma) \leftarrow$ **TJL.Setup**$(\lambda)$ and $(\perp, \perp, N_1, H_1) \leftarrow$ **JL.Setup**$(\lambda)$ s.t. $N_0 \geq 2 \cdot N_1 + \log_2(n)$ and set $pp = (pp^{KA}, N_0, N_1, H_0, H_1, \tau_0, \sigma, t, n, d, R)$
**Prerequisites:** Each client $u \in \mathcal{U}$ generates a key pair $(c_u^{PK}, c_u^{SK}) \leftarrow$ **KA.gen**$(pp^{KA})$ and registers $c_u^{PK}$ to Server or to a PKI

**Setup - Key Setup:**

Client $u \in \mathcal{U}^{(\tau)}$: // Generate **TJL** key and secret share

(1) $\forall v \in \mathcal{U}^{(\tau)} \setminus \{u\}$, $c_{u,v} \leftarrow$ **KA.agree**$(pp^{KA}, c_u^{SK}, c_v^{PK})$. *// Establish pairwise channel keys with each client*
(2) $sk_u \xleftarrow{R} \mathbb{Z}_{N_0^2}$. *// Generate TJL secret key*
(3) $\{(v, [sk_u]_v)\}_{v\in\mathcal{U}^{(\tau)}} \leftarrow$ **TJL.SKShare**$(sk_u, t, \mathcal{U}^{(\tau)})$. *// Generate t-out-of-n shares*
(4) $\forall v \in \mathcal{U}^{(\tau)} \setminus \{u\}$, $\epsilon_{u,v} \leftarrow$ **AE.enc**$(c_{u,v}, u \,||\, v \,||\, [sk_u]_v)$. *// Encrypt each share with the corresponding public key*
(5) Send $\{(u, v, \epsilon_{u,v})\}_{v\in\mathcal{U}^{(\tau)}}$ to *Server*.

Server: *// Collect encrypted shares of TJL keys and forward them to destined clients*

(1) Collect $\{(u, v, \epsilon_{u,v})\}_{v\in\mathcal{U}^{(\tau)}}$.
(2) $\forall v \in \mathcal{U}^{(\tau)} \setminus \{u\}$, send $\{(u, v, \epsilon_{u,v})\}_{u\in\mathcal{U}^{(\tau)}}$.

Client $u \in \mathcal{U}^{(\tau)}$: *// Decrypt the received shares*

(1) $\forall v \in \mathcal{U}^{(\tau)} \setminus \{u\}$, $[sk_v]_u \leftarrow$ **AE.dec**$(c_{u,v}, v \,||\, u \,||\, \epsilon_{v,u})$. *// Decrypt each share with the corresponding public key*

**Online - Protection (step $\tau$):**

Client $u \in \mathcal{U}^{(\tau)}$: *// Protect the private input using JL key, the per-round JL secret key using TJL key, and send them to the server*

(1) $sk_{u,\tau} \xleftarrow{R} \mathbb{Z}_{N_1^2}$. *// Generate the per-round JL secret key*
(2) $\vec{y}_{u,\tau} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau}, \tau_0, \vec{x}_{u,\tau})$. *// Protect private input $\vec{x}_{u,\tau} \in \mathbb{Z}_R^d$ using JL*
(3) $\langle sk_{u,\tau}\rangle \leftarrow$ **TJL.Protect**$(pp, sk_u, \tau, sk_{u,\tau})$. *// Protect the per-round JL secret key $sk_{u,\tau} \in \mathbb{Z}_{N_1^2}$ using TJL*
(4) Send $\vec{y}_{u,\tau}$ and $\langle sk_{u,\tau}\rangle$ to *Server*.

Server: *// If the number of online clients ($\mathcal{U}_{on}^{(\tau)} \subseteq \mathcal{U}^{(\tau)}$) is less than t, abort; otherwise, collect the protected secret keys, the protected inputs, and broadcast $\mathcal{U}_{on}^{(\tau)}$ to all clients*

(1) Collect $\{\langle sk_{u,\tau}\rangle\}_{\forall u\in\mathcal{U}_{on}^{(\tau)}}$ and $\{\vec{y}_{u,\tau}\}_{\forall u\in\mathcal{U}_{on}^{(\tau)}}$.
(2) If $|\mathcal{U}_{on}^{(\tau)}| < t$, abort; otherwise broadcast $\mathcal{U}_{on}^{(\tau)}$.

**Online - Consistency Check (step $\tau$):** *// See Figure 4 (Consistency Check) of [3]*
**Online - Reconstruction (step $\tau$):**

*Client* $u \in \mathcal{U}_{on}^{(\tau)}$: *// Compute the share of the per-round JL server key, and send it to the server*

(1) $[\langle sk'_{0,\tau}\rangle]_u \leftarrow$ **TJL.ShareProtect**$(pp, \{[sk_v]_u\}_{v\in\mathcal{U}_{on}^{(\tau)}}, \tau)$.
(2) Send $[\langle sk'_{0,\tau}\rangle]_u$ to *Server*.

Server: *// If the number of honest clients ($\mathcal{U}_{shares}^{(\tau)} \subseteq \mathcal{U}_{on}^{(\tau)}$) is less than t abort; otherwise collect t shares, reconstruct the per-round JL server key and complete the aggregation*

(1) Collect $\{[\langle sk'_{0,\tau}\rangle]_u\}_{u\in\mathcal{U}_{shares}^{(\tau)}}$.
(2) If $|\mathcal{U}_{shares}^{(\tau)}| < t$, abort; otherwise, proceed.
(3) $sk'_{0,\tau} \leftarrow$ **TJL.ShareCombine**$(\{[\langle sk'_{0,\tau}\rangle]_u\}_{u\in\mathcal{U}_{shares}^{(\tau)}}, t)$. *// Reconstruct the zero-scalar value of the online clients*
(4) $sk_{0,\tau} \leftarrow$ **TJL.Agg**$(pp, 0, \tau, \{sk_{u,\tau}\}_{u\in\mathcal{U}_{on}^{(\tau)}}, sk'_{0,\tau})$. *// Reconstruct the per-round JL server key*
(5) $\vec{x}_\tau \leftarrow$ **JL.Agg**$(pp, -sk_{0,\tau}, \tau_0, \{\vec{y}_{u,\tau}\}_{u\in\mathcal{U}_{on}^{(\tau)}})$. *// Complete the aggregation*

**Figure 1: Eagle in SyncFL.**

reconstructed. Each client protects its private input using a freshly generated per-round **JL** key and this key is then protected with the **TJL** key. Thanks to this approach, **Eagle** exhibits a quasi-linear computation cost at the client. The solution is defined in two phases: the setup phase during which clients first register to the server and receive their keying material, and the online phase during which aggregation occurs.

## 4.1 Description

The protocol is depicted in Figure 1. It starts with the setup phase, where each FL client first generates a pair of secret and public keys and transmits its public key to the FL server who then broadcasts them to all FL clients together with the public parameters $pp$. Delegating the public parameter generation to a TD is common in other existing works*. At the *Key Setup* step, each client independently computes $t$ out of $n$ shares of its secret key $sk_u$ using **TJL.SKShare**. Subsequently, similar to **FTSA** [16], these shares are one-by-one sent to the appropriate clients, via the server, through authenticated encrypted (**AE**) channels. The online phase, is broken down into three steps for each round:

(1) At the *protection step*, online clients generate one per-round **JL** secret key $sk_{u,\tau}$ that is then used to protect their private input vectors $\vec{x}_{u,\tau}$ using **JL.Protect** at round $\tau$, such that the protection with **JL** uses a fixed $\tau = \tau_0$. This key is further protected using **TJL.Protect** and all this information is sent to the FL server. The server gathers both the protected inputs (vectors) and the protected per-round clients' keys (scalars).

(2) The *consistency check step* is the same as for **SecAgg** [3].

(3) At the *reconstruction step*, the clients receive the list of the online clients $\mathcal{U}_{on}^{(\tau)}$. Their goal is to help compute/reconstruct the per-round aggregation key for the server in order to have access to the actual sum of private inputs in plaintext. This aggregation key basically consists of the sum of the per-round keys of online clients that will be reconstructed with the collaboration of at least $t$ online clients using **TJL.ShareCombine**. Then, the actual model parameters $\vec{x}_\tau$ can be computed using **JL.Agg**.

## 4.2 Security Analysis (Sketch)

We briefly analyse the security of **Eagle** by following the approach given in [16], hybrid security proof is provided in Appendix A.

• In the **honest-but-curious** model, we assume that the server correctly follows the protocol but can collude with (or corrupts) up to $n - t$ clients. Let $\mathcal{U}_{corr}$ be the set of corrupted clients and $C = \mathcal{U}_{corr} \cup \mathcal{S}$ where $\mathcal{S}$ 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 $t$ (i.e., $|\mathcal{U}_{corr}| < t$). Based on that, the minimum number of honest clients $t$ should be strictly larger than half of the number of clients in the protocol (i.e., $t > \frac{n}{2}$). Hence the protocol can recover from up to $\frac{n}{2} - 1$ client failures. The security of the **TJL** ensures that parties in $C$ cannot distinguish the protected temporary **JL** key of an honest client $\langle sk_{u,\tau} \rangle$ from random values. It also ensures that if parties in $C$ have access to at most $t - 1$ shares of $sk_u$ (i.e., $|\mathcal{U}_{corr}| < t$), then, they cannot distinguish the shares held by the honest clients

*Note that the generation of public parameters $pp$ depends on the existence of a TD. Nevertheless, there exist methods in the decentralized setting [5, 21, 31].

**Parties:** Server and all clients in $\mathcal{U}$, such that $|\mathcal{U}| = n_{tot}$, and clients in the buffer $\mathcal{U}_{on}$, such that $|\mathcal{U}_{on}| = n$

**Public Parameters:** Generate the public parameters $pp^{KA} \leftarrow$ **KA.Param**$(\lambda)$ and $(\perp, \perp, N, H, \perp) \leftarrow$ **JL.Setup**$(\lambda)$ and set $pp = (pp^{KA}, N, H, \tau_0, t, n_{tot}, d, \mathbb{F}_p, R)$

**Prerequisites:** Each client $u \in \mathcal{U}$ generates a key pair $(c_u^{PK}, c_u^{SK}) \leftarrow$ **KA.gen**$(pp^{KA})$ and registers $c_u^{PK}$ to Server or to a PKI

**Setup - Key Setup:**

Client $u \in \mathcal{U}$:

(1) $\forall v \in \mathcal{U} \setminus \{u\}$, $c_{u,v} \leftarrow$ **KA.agree**$(pp^{KA}, c_u^{SK}, c_v^{PK})$. *// Establish pairwise channel keys with each client*

**Online - Protection:**

Client $u$: *// Protect private input using* **JL** *key, secret share per-client-round* **JL** *secret key*

(1) $sk_{u,\tau_u} \xleftarrow{R} \mathbb{Z}_{N^2}$. *// Generate* **JL** *secret key for round* $\tau_u$
(2) $\vec{y}_{u,\tau_u} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau_u}, \tau_0, \vec{x}_{u,\tau_u})$. *// Protect private input* $\vec{x}_{u,\tau_u}$ *using* **JL** *key*
(3) $\{(v, [sk_{u,\tau_u}]_v)\}_{\forall v \in \mathcal{U}} \leftarrow$ **SS.Share**$(sk_{u,\tau_u}, t, \mathcal{U})$. *// Generate shares of the per-client* **JL** *secret key for round* $\tau_u$
(4) $v \in \mathcal{U} \setminus \{u\}$, $\epsilon_{u,v} \leftarrow$ **AE.enc**$(c_{u,v}, u \,||\, v \,||\, [sk_{u,\tau_u}]_v)$. *// Encrypt each share with the corresponding public key*
(5) Send $\vec{y}_{u,\tau_u}$ and $\{(u, v, \epsilon_{u,v})\}_{u \in \mathcal{U}}$ to *Server.*

Server: *// If the number of clients is less than t, abort; otherwise, collect protected inputs, encrypted shares of secret keys, forward encrypted shares to destined clients and broadcast* $\mathcal{U}_{on}$

(1) Collect $\{\vec{y}_{u,\tau_u}\}_{u \in \mathcal{U}_{on}}$ and $\{(u, v, \epsilon_{u,v})\}_{u \in \mathcal{U}_{on}}$.
(2) If $|\mathcal{U}_{on}| < t$, abort; otherwise, broadcast $\mathcal{U}_{on}$ and $\forall v \in \mathcal{U}_{on}$ send $\{(u, v, \epsilon_{u,v})\}_{u \in \mathcal{U}_{on}}$.

**Online - Consistency Check:** *// See Figure 4 (Consistency Check) of [3]*

**Online - Reconstruction:**

Client $u$: *// Compute the share of the* **JL** *aggregation key*

(1) $\forall v \in \mathcal{U}_{on} \setminus \{u\}$, $[sk_{v,\tau_v}]_u \leftarrow$ **AE.dec**$(c_{u,v}, v \,||\, u \,||\, \epsilon_{v,u})$. *// Decrypt each share*
(2) $[sk_0]_u \leftarrow \sum_{\forall v \in \mathcal{U}_{on}} [sk_{v,\tau_v}]_u$. *// Compute share of per-round aggregation key*
(3) Send $[sk_0]_u$ to *Server.*

Server: *// If the number of honest clients is less than t, abort; otherwise, collect t shares, reconstruct the per-client-round* **JL** *server key and complete the aggregation*

(1) Collect $\{[sk_0]_u\}_{u \in \mathcal{U}_{shares}}$.
(2) If $|\mathcal{U}_{shares}| < t$, abort; otherwise, proceed.
(3) $sk_0 \leftarrow$ **SS.Recon**$(\{[sk_0]_v\}_{v \in \mathcal{U}_{shares}}, t)$. *// Reconstruct* **JL** *aggregation key*
(4) $\vec{x} \leftarrow$ **JL.Agg**$(pp, -sk_0, \tau_0, \{\vec{y}_{u,\tau_u}\}_{u \in \mathcal{U}_{on}})$. *// Compute aggregate value*

**Figure 2: Owl in BAsyncFL.**

from random values. Therefore, the view of parties in $C$ at the end of each FL round $\tau$ is computationally indistinguishable from a simulated view. Thus, the server learns nothing more than the sum of the online clients' inputs if $|\mathcal{U}_{on}^{(\tau)}| \geq |\mathcal{U}_{shares}^{(\tau)}| \geq t$ and hence *AO* is ensured.

• In the **active** model, $\mathcal{S}$ can additionally manipulate its inputs to the protocol. The only messages $\mathcal{S}$ distributes, other than the clients' public keys, are the protected shares that are forwarded from and to the clients. $\mathcal{S}$ cannot modify the values of these encrypted shares thanks to the underlying authenticated encryption scheme **AE**. Therefore, $\mathcal{S}$'s power in the protocol is limited to not forwarding some of the shares. This may make clients reach some false conclusions about the set of online clients $\mathcal{U}_{on}^{(\tau)}$. It is important to note that $\mathcal{S}$ can present different views to different clients regarding their online/dropped status. This capability enables $\mathcal{S}$ to easily acquire the individual temporary **JL** key $sk_{u,\tau}$ of a client $u$. More precisely, $\mathcal{S}$ can convince a subset of honest clients that the set of online clients is $\mathcal{U}_{on}^{(\tau)}$ while indicating to another subset that the online clients' set is $\mathcal{U}_{on}^{(\tau)'} = \mathcal{U}_{on}^{(\tau)} \setminus \{u\}$ (i.e., $u$ is dropped). If this occurs, $\mathcal{S}$ can aggregate the protected inputs from $\mathcal{U}_{on}^{(\tau)}$ to derive the per-round aggregation key $sk_\tau$, and also aggregate inputs from $\mathcal{U}_{on}^{(\tau)'}$ to derive $sk'_\tau$, and then calculate $sk_{u,\tau} = sk_\tau - sk'_\tau$. Considering the scenario where $\mathcal{S}$ may collude with $n-t$ corrupted clients, it can obtain $n-t$ shares of $\langle sk_\tau \rangle$ and $\langle sk'_\tau \rangle$ and hence $n-t$ shares of $sk_{u,\tau}$. Furthermore, $\mathcal{S}$ has the ability to convince $\frac{t}{2}$ honest clients that client $u$ is online, and the other $\frac{t}{2}$ honest clients that $u$ is dropped, thereby collecting shares of $\vec{y}_\tau$ and $\vec{y}'_\tau$ respectively. Hence, in total, the server can learn a maximum number of $n-t+\frac{t}{2}$ shares of $\langle sk_\tau \rangle$ and $\langle sk'_\tau \rangle$. Therefore, to prevent such attacks and still ensure *AO*, we additionally 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 in the active model.

To conclude, if the server operates under an **honest-but-curious** model, selecting $t > \frac{n}{2}$ ensures security. However, if the server **actively** manipulates protocol messages, the threshold should be set to $t > \frac{2n}{3}$. Additional details about the threat model and the rationale behind these thresholds can be found in [3] (Sections 6.1 and 6.2). Note that we achieve the same threshold values as those in **SecAgg** [3] and **FTSA** [16].

• A new type of attack, called model inconsistency attack, launched by an active aggregator is defined and studied in [23]. This attack consists of a malicious server sending carefully crafted models to specific clients instead of the actual global model parameters, with the aim of extracting private clients' parameters. **Eagle** can easily prevent such attacks by adopting the same approach proposed in [23]. In more details, using the hash of the global model to set the server's value $\tau_0$, which needs to be the same for all clients, allows to overcome the aforementioned attack.

## 5 OWL IN ASYNCFL

In the asynchronous setting, we propose **Owl**, defined with a setup phase and an online phase. As opposed to the case in SyncFL, in the context of AsyncFL, clients cannot be expected to be synchronized with respect to the same round $\tau$. Consequently, by design, the **TJL** scheme cannot be used directly. To counter this problem, one common round $\tau_0$ is defined per client $u$ and each client uses **JL.Protect** with its own per-round key $sk_{u,\tau_u}$. On the other hand, the FL server also defines its own round $\tau_0$ (which, in fact, never changes) and is still able to aggregate all values thanks to the use of **JL.Agg** with $\tau_0$ to get the per-round aggregate key and to further obtain the aggregate model.

### 5.1 Description

The protocol is depicted in Figure 2. It starts with the setup phase, similar to **Eagle**, where each FL client generates a pair of secret and public keys and transmits its public key to the FL server who then broadcasts it to all FL clients, together with the public parameters *pp*. The online phase, consists of three steps:

(1) During the *protection step*, each online client $u$ generates one **JL** secret key at round $\tau_u$, denoted as $sk_{u,\tau_u}$. The client then computes $n$ shares of this secret key such that any $t$ shares can reconstruct it, using the **SS** scheme. Subsequently, the private input $\vec{x}_{u,\tau_u}$ (vector) is protected using **JL.Protect** which takes as inputs $sk_{u,\tau_u}$ and a fixed $\tau_0$ defined by the server. The server collects both the protected inputs and the encrypted shares of the protection key from the online clients.

(2) The *consistency check step* is the same as for **SecAgg** [3].

(3) At the *reconstruction step*, each client $u$ receives the encrypted shares of each other client's protection key and computes the share of the server's **JL** aggregation key $\sum_{\forall v \in \mathcal{U}_{on}} [sk_{v,\tau_v}]_u$. This global share is forwarded to the server. The server should receive at least $t$ shares to reconstruct the aggregation key $sk_0$. Finally, the server aggregates the inputs of the online clients using **JL.Agg**.

### 5.2 Security Analysis (Sketch)

We briefly analyse the security of **Owl** by following the approach given in [16], hybrid security proof is provided in Appendix A.

• In the **honest-but-curious** model, the **JL** scheme ensures that the server together with clients in $C$ cannot distinguish protected inputs $\vec{y}_{u,\tau_u}$ from random values. Furthermore, the **SS** scheme ensures that if parties in $C$ have access to less than $t-1$ shares of the client's secret key $sk_{u,\tau_u}$ (i.e. $|\mathcal{U}_{corr}| < t$), then they cannot distinguish the shares held by the honest clients from random values. Therefore, the view of clients in $C$ is computationally indistinguishable from a simulated view. Thus, the server learns nothing more than the sum of the online clients' inputs if $|\mathcal{U}_{on}^{(\tau)}| \geq |\mathcal{U}_{shares}^{(\tau)}| \geq t$.

• When $\mathcal{S}$ is an **active** adversary, it can try to convince a subset of honest clients that the set of online clients is $\mathcal{U}_{on}$ while indicating to another subset that the set of online clients is $\mathcal{U}'_{on} = \mathcal{U}_{on} \setminus \{u\}$ for a client $u$. If this occurs, $\mathcal{S}$ can reconstruct $sk_0$ and $sk'_0$ and then, it can compute $sk_{u,\tau_u} = sk_0 - sk'_0$. Because we assume that there are $n-t$ corrupted clients, $\mathcal{S}$ can obtain $n-t$ shares of $sk_0$ and $sk'_0$ respectively. Furthermore, $\mathcal{S}$ has the ability to convince $\frac{t}{2}$ honest clients that the client $u$ is online, and the other $\frac{t}{2}$ honest clients that $u$ is dropped, thereby collecting shares of $sk_0$ and $sk'_0$ respectively. Therefore, to ensure *AO*, we require that $n-t+\frac{t}{2} < t \implies t > \frac{2n}{3}$.

To conclude, if $\mathcal{S}$ is **honest-but-curious**, selecting $t > \frac{n}{2}$ ensures security. However, if $\mathcal{S}$ **actively** manipulates protocol messages, the threshold should be $t > \frac{2n}{3}$. We get the same threshold values obtained in [30].

• As mentioned in [23], defenses against model inconsistency attacks in AsyncFL settings are a difficult task and no one has yet proposed a potential solution. Consequently, those attacks are out of scope for **Owl**.

## 6 RELATED WORK

*Secure Aggregation for SyncFL.* Bonawitz et al. [3] propose **SecAgg**, a fault-tolerant SA approach that employs secure masking. Each pair of clients creates a shared mask through a key agreement scheme and uses this mask to protect clients' inputs. Additionally, before this protection, clients also combine their input data with another blinding mask. The purpose of this blinding mask is to prevent any active, malicious server from discovering one individual input at the reconstruction step. To address the potential issue of client dropout, the protocol implements secret sharing: the clients secretly share their respective shared masks and blinding masks. Then, the server computes the sum of the masked inputs and further recovers the shared masks of the failed clients and the blinding masks of the online clients, thereby completing the aggregation. Compared to

**SecAgg**, **Eagle** does not use any blinding mask and consequently is more efficient in terms of computation and communication costs.

Mansouri et al. [16] develop a fault-tolerant SA solution called **FTSA**, which uses the **TJL** scheme to protect client inputs and reconstruct the aggregate in case of client failures, reducing the online communication rounds from 4 to 3 rounds compared to **SecAgg**. To address the issue of potential client dropouts, the clients secretly share their respective secret keys using the **ISS** scheme. Similar to **SecAgg**, **FTSA** also uses blinding masks for clients' inputs which once again increases the communication cost compared to our protocol. Furthermore, **FTSA** cannot directly support *client selection* as the non-selected clients would be considered as failed clients and this would significantly increase the computation and communication overhead. Our solution instead only depends on the number of online client.

Ma et al. [15] introduce **Flamingo**, a fault-tolerant SA protocol employing secure masking. The authors construct connected graphs of clients (as opposed to fully connected clients) such that the shared mask is created among the connected clients. **Flamingo** introduces the concept of *decryptors* to help the server reconstruct the aggregate, as opposed to distribute the reconstruction to all clients. Its functioning is as follows: each pair of connected clients generates a shared seed through key agreement and creates a shared pairwise mask using a Pseudo-Random Function (PRF). Moreover, a blinding mask is created. To address the potential issue of client dropout, the protocol implements Threshold ElGamal asymmetric encryption (TEG). In **Flamingo**, clients encrypt the pairwise mask using TEG, while the blinding mask is secretly shared with the decryptors. If a client fails, the server asks the decryptors to reconstruct the pairwise mask using the partial decryption of TEG. Otherwise, they reconstruct the blinded mask for the online clients. Consequently, **Flamingo** incurs three client-server trips. Similar to previously explained protocols, the computational complexity increases with the use of the blinding mask. Additionally, its complexity scales with the number of dropped clients, impacting the number of connected clients in the sparse graph. Furthermore, dropped clients have a negative impact during the reconstruction led by the decryptors, as the more they drop, the more pairwise masks the decryptors and the server have to reconstruct.

Bell et al. [1] enhance the scalability of **SecAgg** [3]. Their method does not require the clients to secretly share secret keys with every other client to ensure resilience against dropouts. Instead, the authors build connected graphs of clients (as opposed to fully connected clients) in which SA is exclusively carried out among the connected clients. We did not provide a detailed experimental comparison, as [15] already demonstrated that [1] requires 6 online communication rounds compared to 3 for **Flamingo** and **Eagle**.

Several protocols provide slight improvements either on the computation cost [9] or on the communication cost [29]. We do not extensively compare **Eagle** with these solutions, mainly because they do not offer the same privacy guarantees.

A recent work [14] proposes a SA method, called **Lerna**, for a large number of clients and which shows a similar DCR construction to **Eagle**. However, their work focuses only on a large number of clients, without considering dropped clients/stragglers and AsyncFL settings. Therefore, we choose to not include **Lerna** in our comparisons.

*Secure Aggregation for AsyncFL.* So et al. [30] propose a SA method, called **LightSecAgg**, that is designed to be compatible with AsyncFL and is the closest solution to **Owl**. In **LightSecAgg**, each client independently generates a random mask which is further secretly shared (using Lagrange code computing [34]) among other clients. At the protection step, each client adds the mask to protect its local model and sends the masked model to the server. At the reconstruction step, the server can reconstruct and cancel out the aggregated masks of the online clients through one-shot decoding. This decoding is performed using the aggregated shared masks received in a second round of communication. **Owl** offers better scalability mainly because, instead of secretly sharing the random mask, it only requires the secret sharing of a single value, the client's **JL** secret key. This optimization significantly reduces both the runtime and communication costs associated with the protocol.

Other protocols, such as **FedBuff** [20], rely on the use of a Trusted Execution Environment (TEE). While such solutions may be more efficient, such a memory-constrained technology cannot be assumed available in all FL settings.

# 7 COMPLEXITY ANALYSIS

## 7.1 Eagle

We evaluate the computation and communication costs of **Eagle** and we compare them with **SecAgg** [3] and **FTSA** [16]. Table 1 on page 2 summarizes this study. Note that our analysis considers the size of the secret shares since this metric has a non-negligible impact on the *reconstruction step*.

- *Client computation:* Firstly, at the *protection step*, the client protects its $d$-size input $\vec{x}_{u,\tau}$ which results in a cost worth $O(d)$. Then, at the *reconstruction step*, the client executes **TJL.ShareProtect** which consists of: *(i)* computing the sum of the secret shares of online clients which requires a computational cost of $O(n)$ as opposed to $O(n^2)$ for **FTSA** and **SecAgg**, mainly because both the latter compute the secret shares of the blinding mask; *(ii)* protecting the zero-*scalar* value using this sum of secret shares through **ISS**, and hence incurring an overhead of $O(n\log(n))$, as opposed to $O(n\log(n)d)$ for **FTSA**. Thus, the overall computation cost for this step is $O(n\log(n)+d)$, which is a quasi-linear complexity.
- *Client communication:* There are two communication rounds. Firstly, at the *protection step*, the client sends its protected input $\vec{y}_{u,\tau}$ to the server, which has a size of $O(d)$. Then, at the *reconstruction step*, the client receives the information on other online clients which is of size $O(n)$, and sends one zero-*scalar* value protected with the combination of their shares, namely $sk'_{0,\tau}$, to the server, which has a constant size $O(1)$. Hence, the communication cost at the client is $O(n+d)$ which is asymptotically the same as **FTSA** and **SecAgg**. Nevertheless, in **FTSA**, during the reconstruction step, a zero-*vector* value is sent which has the same size as of the protected input, that is $O(d)$. When client dropouts occur, our solution would outperform **FTSA** mainly because **Eagle** does not depend on the number of client failures. This is also experimentally studied and evaluated in Section 8.
- *Server computation:* The server performs two main operations: *(i)* at the *reconstruction step*, it reconstructs the protected zero-*scalar* value $sk'_{0,\tau}$ from the $t$ shares of the online clients, requiring a computation cost of $O(n^2+n)$ (from Lagrange coefficients); *(ii)*

the server aggregates the protected values and unmasks the result, which requires a computation cost of $O(nd)$. The overall cost is the same as in **FTSA**, worth $O(n^2 + nd)$. As at the clients, the computation cost at the server is not impacted by dropouts. On the other side, in **FTSA** and **SecAgg**, the server needs to reconstruct the blinding masks of dropped clients. In particular, in **FTSA**, the computation of the protected zero-*vector* value of the dropped clients incurs a cost of $O(n^2 + nd)$, and in **SecAgg**, the masks of the dropped clients are reconstructed with a complexity $O(n^2 d)$. This is also experimentally studied and evaluated in Section 8.

• *Server communication:* Similar to **SecAgg** and **FTSA**, since the message exchanges in the protocol only occur between the server and the clients, the server's communication cost is equal to $n$ times each client's communication cost.

| | **Flamingo [15]** | **Eagle** |
|---|---|---|
| **Client Comp.** | Regular client: $O(k^2 + ad)$<br>Decryptor: $O(\delta an + (1-\delta)n)$ | Regular client: $O(d)$<br>Decryptor: $O((1-\delta)n + k\log k)$ |
| **Client Comm.** | Regular client: $O(a + k + d)$<br>Decryptor: $O(\delta an + (1-\delta)n)$ | Regular client: $O(d)$<br>Decryptor: $O(n)$ |

**Table 3: Complexity analysis for one SyncFL round with decryptors ($n$: number of clients, $d$: input dimension; $k$: number of decryptors; $\delta$: dropout rate; $a$: upper bound on the number of neighbors per client).**

***Eagle*** *with decryptors.* We conduct a comparative complexity analysis of **Eagle** when decryptors are involved, and compare the online phase against **Flamingo** [15]. We distinguish regular clients (who contribute to the global model) from decryptors (who help the server obtain this global model). Let the dimension of the model be $d$, the dropout rate be $\delta$, the number of neighbors for a given client be $a$, the number of decryptors be $k$ and the number of regular clients be $n$. Note that the number of neighbors $a$ is determined by the dropout rate $\delta$, as shown in [15] (see Appendix C). The results are summarized in Table 3.

• *Client Computation:* For regular clients in **Eagle**, the primary computational cost is attributed to the *protection step*, worth $O(d)$. In the case of **Flamingo**, the protection cost is $O(ad)$. This cost involves the secret sharing of the blinding mask with $k$ decryptors, incurring an expense of $O(k^2)$, and the TEG encryption of the pairwise masks worth $O(a)$.

For decryptors in **Eagle**, the computational requirements resemble those without decryptors, with the different being the size of the **TJL** secret key share, which is $k \log k$. Consequently, the cost associated with protecting the zero-scalar value becomes $O(k \log k)$, resulting in a total cost of $O(n+k \log k)$. In the context of **Flamingo**, the overhead incurred by decryptors is proportional to the fraction of dropped clients $\delta$, since decryptors partially decrypt $\delta an$ TEG ciphertexts and decrypt $(1-\delta)n$ secret shares.

• *Client Communication:* For regular clients in **Eagle**, the communication cost worth $O(d)$ depends on the protected input size $d$. In the case of **Flamingo**, in addition to the protected input, the communication also involves sending $a$ TEG ciphertexts and $k$ encrypted secret shares of the blinding mask.

For decryptors in **Eagle**, the only required information is the set of online clients, which has to be reconstructed, incurring a cost of $O((1-\delta)n)$. In the context of **Flamingo**, the decryptors receive from the server $\delta an$ TEG ciphertexts and $(1-\delta)n$ encrypted blinding masks, costing $O(\delta an + (1-\delta)n)$.

• *Server Computation:* During the *reconstruction step* in **Eagle**, the server reconstructs the protected zero-scalar value using $t$ shares from the decryptors, requiring a computational cost of $O(k^2 + k)$ (due to Lagrange coefficients' computation). Subsequently, the server aggregates the protected values and unmasks the result, which requires a computational cost of $O(nd)$.

In **Flamingo**, the *reconstruction step* requires the cost of the Lagrange coefficients' computation, worth $O(k^2)$, in addition to the reconstruction of $(1-\delta)n$ blinding masks, incurring a cost of $O((1-\delta)nk)$. The reconstruction of the pairwise masks implies a cost of $O(\delta ank)$, and then the aggregation and unmasking cost $O(k^2 + \delta ank + (1-\delta)nk + nd)$.

• *Server Communication:* In both protocols, the communication cost is equal to $n$ times the client's communication cost.

## 7.2 Owl

We evaluate our protocol **Owl** tailored for AsyncFL and compare its costs with **LightSecAgg** [30]. Table 1 on page 2 summarizes our study and shows that our solution outperforms **LightSecAgg**. We set the complexity of polynomial evaluation and interpolation as $O(n^2)$ for all solutions. Note that this complexity can be reduced to $O(n \log n)$ as pointed out in [30] and acknowledged in [27].

• *Client computation:* At the *protection step*, the client generates $t$ out of $n$ shares of the secret key $sk_{u,\tau_u}$, which requires a computation cost of $O(n^2)$. Also, the client protects its message $\vec{x}_{u,\tau_u}$ using the secret key $sk_{u,\tau_u}$, which requires a computation cost of $O(d)$. Finally, at *reconstruction step*, the client computes the sum of the secret key shares of other online clients, which requires a computation cost of $O(n)$. This cost is better than in **LightSecAgg** , which is $O(n^2 \frac{d}{(1-\delta)n-t} + d)$, mainly due to their underlying encoding [34].

• *Client communication:* At the *protection step*, the client sends $O(n)$ shares of its secret key $sk_{u,\tau_u}$ and receives $O(n)$ shares in return. The client further sends the encrypted input $\vec{y}_{u,\tau_u}$ to the server, which is of size $O(d)$. Finally, at the *reconstruction step*, the client sends its share of the server's secret key $[sk_0]_u$ which has a size of $O(1)$. The total cost, worth $O(n + d)$, is better than $O(n \frac{d}{(1-\delta)n-t} + d)$ from **LightSecAgg**.

• *Server computation:* At the *reconstruction step*, the server constructs the server key $sk_0$ from its $t$ shares, which requires a computation cost of $O(n^2)$. Additionally, the server aggregates the ciphertexts received from each client and unmasks the result, which requires a computation cost of $O(nd)$.

• *Server communication:* Similar to **Eagle**, since the message exchanges in the protocol only occur between the server and the clients, the server's communication cost is equal to $n$ times each client's communication cost.

# 8 EXPERIMENTAL STUDY

We also conduct an experimental study of the performance of **Eagle** and **Owl**, with respect to the number of selected clients and buffer size respectively, the size of the machine learning model, while considering realistic dropout rates and over selection. We also study

| N. Clients | Dim. | Drop. | Client | | | | | | Server | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Wall-clock running time (s) | | | Total data transfer (sent/received) (MB) | | | Wall-clock running time (s) | | |
| | | | **SecAgg** [3] | **FTSA** [16] | **Eagle** | **SecAgg** [3] | **FTSA** [16] | **Eagle** | **SecAgg** [3] | **FTSA** [16] | **Eagle** |
| 512 | $10^5$ | 0.0 | 35.32 | 7.81 | **7.15** | **0.49** | 0.71 | 0.64 | 496.12 | 52.08 | **18.08** |
| | | 0.1 | 35.39 | 48.77 | **7.14** | **0.49** | 1.35 | 0.64 | 2167.81 | 6516.23 | **17.61** |
| | | 0.3 | 35.48 | 48.79 | **7.13** | **0.49** | 1.34 | 0.64 | 5527.75 | 5020.67 | **16.81** |
| | $10^6$ | 0.0 | 336.23 | 72.99 | **70.86** | **3.30** | 6.47 | 6.40 | 2376.32 | 390.31 | **120.21** |
| | | 0.1 | 336.91 | 404.82* | **71.11** | **3.30** | 12.87* | 6.40 | 19994.62 | > 7d. | **116.59** |
| | | 0.3 | 340.87 | 403.86* | **70.71** | **3.30** | 12.87* | 6.40 | 52499.26 | > 7d. | **107.28** |
| 1024 | $10^5$ | 0.0 | 73.02 | 8.90 | **7.55** | 0.67 | 0.79 | **0.66** | 2904.75 | 187.34 | **45.94** |
| | | 0.1 | 72.67 | 94.21 | **7.36** | 0.67 | 1.45 | **0.66** | 9449.02 | 29534.22 | **45.03** |
| | | 0.3 | 72.61 | 93.79 | **7.40** | 0.68 | 1.43 | **0.66** | 22853.70 | 23470.55 | **43.08** |
| | $10^6$ | 0.0 | 667.20* | 75.46 | **72.75** | **3.60*** | 6.70 | 6.57 | > 7d. | 861.19 | **193.16** |
| | | 0.1 | 655.43* | 739.85* | **72.81** | **3.60*** | 13.20* | 6.57 | > 7d. | > 7d. | **185.30** |
| | | 0.3 | 658.34* | 743.77* | **72.49** | **3.60*** | 13.20* | 6.57 | > 7d. | > 7d. | **164.72** |

**Table 4: Computation and communication costs per client and computation costs for the server, for one SyncFL round. Comparison with SecAgg and FTSA. Values highlighted with "*" are the result of an estimation since the underlying experiments took more than seven days and hence were aborted.**

| N. Clients | Dim. | Drop. | Regular client / Decryptor | | | | Server | |
|---|---|---|---|---|---|---|---|---|
| | | | Wall-clock running time (s) | | Total data transfer (sent/received) (MB) | | Wall-clock running time (s) | |
| | | | **Flamingo** [15] | **Eagle** | **Flamingo** [15] | **Eagle** | **Flamingo** [15] | **Eagle** |
| 512 | $10^5$ | 0.0 | **0.30** / 0.06 | 7.13 / **0.02** | **0.31** / 0.04 | 0.64 / **0.01** | 239.91 | **11.47** |
| | | 0.1 | **6.44** / 3.67 | 7.14 / **0.02** | **0.31** / 0.38 | 0.64 / **0.01** | 742.09 | **11.00** |
| | | 0.3 | 15.53 / 20.33 | **7.09** / **0.02** | **0.31** / 1.95 | 0.64 / **0.01** | 2936.63 | **10.15** |
| | $10^6$ | 0.0 | **2.50** / 0.11 | 70.57 / **0.02** | **3.12** / 0.04 | 0.64 / **0.01** | 2049.78 | **113.83** |
| | | 0.1 | **64.78** / 3.84 | 70.65 / **0.02** | **3.12** / 0.38 | 0.64 / **0.01** | 4799.36 | **109.76** |
| | | 0.3 | 156.50 / 20.69 | **70.93** / **0.02** | **3.12** / 1.95 | 0.64 / **0.01** | 18720.37 | **101.48** |
| 1024 | $10^5$ | 0.0 | **0.07** / 0.13 | 7.33 / **0.02** | **0.33** / 0.08 | 0.64 / **0.01** | 66.12 | **16.38** |
| | | 0.1 | 12.13 / 14.90 | **7.30** / **0.02** | **0.33** / 1.44 | 0.64 / **0.01** | 1807.76 | **15.48** |
| | | 0.3 | 28.54 / 79.57 | **7.41** / **0.02** | **0.33** / 7.49 | 0.64 / **0.01** | 10381.25 | **13.54** |
| | $10^6$ | 0.0 | **4.38** / 0.35 | 72.46 / **0.02** | 3.25 / 0.08 | 0.64 / **0.01** | 683.48 | **163.41** |
| | | 0.1 | 129.91 / 14.94 | **72.44** / **0.02** | 3.25 / 0.08 | 0.64 / **0.01** | 12703.08 | **154.31** |
| | | 0.3 | 300.20 / 78.74 | **72.26** / **0.02** | 3.25 / 7.46 | 0.64 / **0.01** | 75420.51 | **135.11** |

**Table 5: Computation and communication costs per regular client and decryptor, and computation costs for the server, for one SyncFL round. Comparison with Flamingo with a number of decryptors set to 60.**

use cases of training a machine learning model for MNIST, CIFAR-10 and Shakespeare datasets.

## 8.1 Experimental Setting

All our implementations use the Python programming language†. Experiments were carried out on a single-threaded process, using a machine equipped with an Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz processor and 126 GB of RAM. For the sake of fair comparison, our solutions along with **SecAgg**, **FTSA**, **Flamingo**, and **LightSecAgg** are implemented using the same building blocks and libraries mentioned in [16] (see Appendix C).

We consider different settings that simulate realistic environments. The number of selected SyncFL clients and the size of the AsyncFL buffer is set to $n = \{512, 1024\}$, and the model size is $d = \{10^5, 10^6\}$. Similar to previous works, the client dropout and over-selection rates are set to $\delta = \{0.0, 0.1, 0.3\}$, and the chosen threshold to $t = \frac{2n}{3}$. We assume that client dropouts happen before the clients send their protected inputs. This is essentially the "worst dropout case", since the server must perform an expensive recovery computation to correctly compute the aggregate. Regarding the packing technique, we adopt the vector encoding approach proposed by [16].

We measure the execution time (i.e. computation cost) and the bandwidth (i.e. communication cost) at both the client and the server sides. The values shown for each experiment are the result of an average of measurements from 5 independent executions. We report performance results of the the setup phase in Figure 3. Below, we detail the performance results of the online phase below.

## 8.2 Eagle

We evaluate the performance of **Eagle** by first comparing it with **SecAgg** and **FTSA** since these three solutions have similar settings. Since **Flamingo** considers the involvement of some *decryptors* who help reconstruct clients' material (see Section 7.1), we conduct additional experiments that simulate these decryptors for **Eagle** and enable a fair comparison with **Flamingo**.

***Eagle** vs. **SecAgg** and **FTSA**.* Table 4 depicts the running time of one FL client as well as the total data transfer for various realistic

†The code of the paper can be found at: GitHub repository.

| N. Clients | Dim. | Drop. | Client | | | | Server | |
|---|---|---|---|---|---|---|---|---|
| | | | Wall-clock running time | | Total data transfer (sent/received) (MB) | | Wall-clock running time | |
| | | | **LightSecAgg** [30] | **Owl** | **LightSecAgg** [30] | **Owl** | **LightSecAgg** [30] | **Owl** |
| 512 | $10^5$ | 0.0 | > 7d | **7.72s** | – | **0.96** | > 7d | **12.85s** |
| | | 0.1 | > 7d | **7.66s** | – | **0.96** | > 7d | **12.13s** |
| | | 0.3 | > 7d | **7.78s** | – | **0.96** | > 7d | **10.77s** |
| | $10^6$ | 0.0 | > 7d | **71.28s** | – | **6.72** | > 7d | **116.60s** |
| | | 0.1 | > 7d | **72.00s** | – | **6.72** | > 7d | **111.80s** |
| | | 0.3 | > 7d | **71.60s** | – | **6.72** | > 7d | **102.58s** |
| 1024 | $10^5$ | 0.0 | > 7d | **12.88s** | – | **1.31** | > 7d | **22.29s** |
| | | 0.1 | > 7d | **9.48s** | – | **1.30** | > 7d | **20.31s** |
| | | 0.3 | > 7d | **9.46s** | – | **1.30** | > 7d | **16.65s** |
| | $10^6$ | 0.0 | > 7d | **74.76s** | – | **7.21** | > 7d | **171.74s** |
| | | 0.1 | > 7d | **75.12s** | – | **7.21** | > 7d | **159.11s** |
| | | 0.3 | > 7d | **74.50s** | – | **7.21** | > 7d | **137.53s** |

**Table 6: Computation and communication costs per client and computation costs for the server, for one AsyncFL round. Comparison with LightSecAgg. "> 7d" denotes experiments with an overall execution taking more than seven days.**

settings. As expected, we observe that the best running time is obtained with **Eagle**, which is independent of the dropout rates. While there is a significant difference with **SecAgg** due to its strong dependence on the number of clients $n$, it is lighter in the case of **FTSA** when there is no dropout, since **Eagle** does not require any sharing of blinding masks. Even if only one client drops, the running time at the client in **FTSA** increases significantly because the reconstruction of dropped clients' inputs is performed over all model parameters and not over a key such as in **Eagle** (see Section 7). When $n = 1024$, $d = 10^6$ and $\delta = 0.1$, **Eagle** is approximately 10× faster than the two other solutions. Regarding the communication cost, when dealing with large client inputs, **SecAgg** exhibits the best communication cost, as also identified in [16]. This is primarily because **FTSA** and **Eagle** use vector encoding, whereas **SecAgg** implements secure masking. On the other hand, our protocol always shows better results compared with **FTSA**. This is achieved thanks to the **TJL** protection of a scalar instead of a vector. In conclusion, on the client's side, we believe that **Eagle** shows its best performance when dropouts would most probably happen.

We have also evaluated the computation cost of the FL server for one SyncFL round, and depict the results in Table 4. We do not show the communication cost as this would correspond to $n$ times the communication cost of one FL client. We observe that the running time of the FL server in **SecAgg** and **FTSA** increases with the dropout rate, while this trend is reversed when it comes to **Eagle**. The reason behind this performance comes from the fact that the number of online clients decreases, and consequently the aggregation time, when the dropout rate increases.

***Eagle** vs. **Flamingo**.* To compare the performance of **Eagle** with **Flamingo**, we have emulated **Flamingo**'s environment and re-implemented **Eagle** accordingly. As detailed in Section 6, **Flamingo** employs a pairwise masking scheme built upon the creation of a random sparse graph and introduces $k$ decryptors, which correspond to special clients helping the server reconstruct the aggregate. We incorporate the concept of decryptors in **Eagle** by involving them during the reconstruction phase. Accordingly, the threshold value $t$ is set to $\frac{2}{3}k$. We have conducted similar experiments as above, with the addition of the value $k = 60$ as in [15].

Table 5 shows our experimental results for regular clients, decryptors and server. As expected, **Flamingo** is the preferred choice when the dropout rate is null. However, as the dropout rate increases ($\delta \geq 0.1$), **Flamingo** becomes more costly in terms of regular client computation. This is mainly due to the increasing number of the clients' neighbors, as detailed in Section 7.1. On the other hand, **Flamingo** is always better in term of communication since its plaintext space is smaller than **Eagle**. Regarding decryptors, **Eagle** is consistently better for both computation and communication, primarily due to the costly reconstruction operations for the pairwise masks in **Flamingo**. To summarize, **Eagle** shows its best performance with $n = 1024$, $d = 10^5$ and a dropout rate exceeding $\delta = 0.1$ when compared with **Flamingo**. Specifically, **Eagle** is ×4 better for computation and ×3 better for communication in the aforementioned scenario.

## 8.3 Owl

We also run the previously described experiments for **Owl** and **LightSecAgg** [30] in the asynchronous setting. In Table 6, we report the wall-clock time and size of the data transferred within a single FL round, for one FL client and one FL server respectively. Firstly, we observe that **Owl** always exhibits better computation time, as

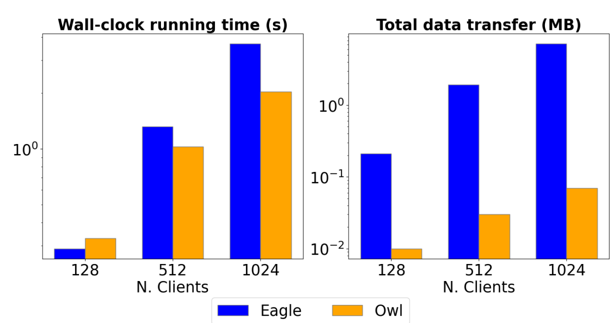


**Figure 3: Computation and communication costs (sent/received) per client during the setup phase of Eagle and Owl.**

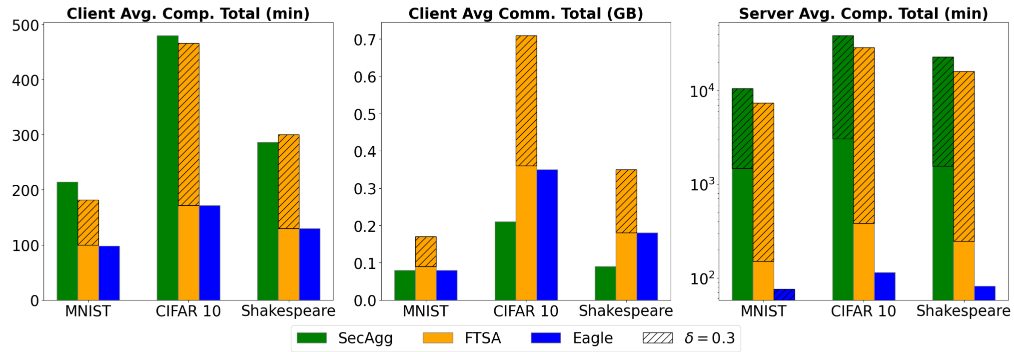


**Figure 4: FL training simulation of Eagle, SecAgg and FTSA ($\delta$ is the dropout rate).**

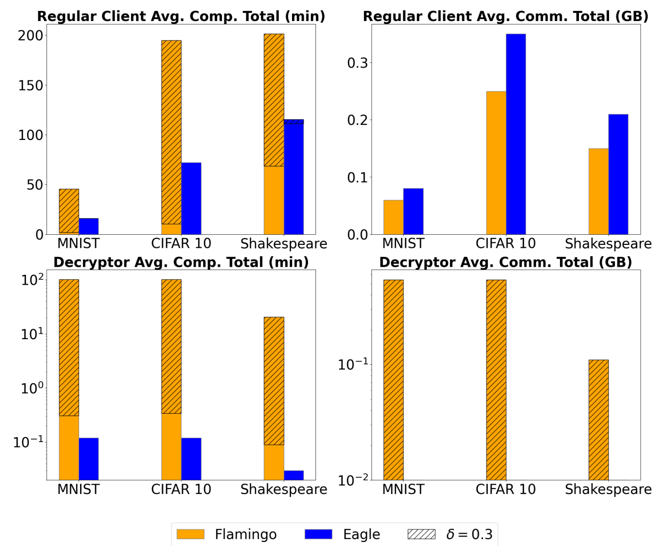


**Figure 5: FL training simulation of Eagle and Flamingo with 60 decryptors ($\delta$ is the dropout rate).**

already identified in the asymptotic analysis (see Section 7.2). In **LightSecAgg**, the computation time increases with the dropout rate, mainly because the reconstruction step is conducted by fewer online clients. Furthermore, the computation and communication costs of **LightSecAgg**, with a model size larger than $10^5$ and a buffer size larger than 512, could not be measured as the execution took more than seven days. To conclude, **Owl** exhibits the best performance in AsyncFL.

## 8.4 Realistic Use Cases

In Figures 4 and 5, we report the results of experiments conducted on three FL tasks, namely MNIST [13] ($d = 61k$ parameters, 0.99 accuracy), CIFAR-10 [12] ($d = 270k$ parameters, 0.83 accuracy), and Shakespeare [4] ($d = 819k$ parameters, 0.56 accuracy). We consider a first scenario without client failure and another scenario with client failure with a dropout rate set to 30%. We first compare **Eagle** against **SecAgg** and **FTSA**. When decryptors are available, we compare **Eagle** against **Flamingo**. Neural network training is performed using Python in PyTorch framework [24] without GPU acceleration. For each online communication round, we consider a timeout of 10 seconds as in [15]. We set $n = 400$ and use SGD as the training algorithm with learning rate $\eta$, number $T$ of FL rounds, batch size $B$, number $E$ of epochs, and number $S$ of samples. More precisely: $(i)$ for MNIST: $T = 300$, $\eta = 0.1$, $B = 32$, $E = 5$ and $S = 150$; $(ii)$ for CIFAR-10: $T = 300$, $\eta = 0.1$, $B = 8$, $E = 4$ and $S = 125$; $(iii)$ for Shakespeare: $T = 60$, $\eta = 0.3$, $B = 8$, $E = 1$ and $S = 2000$. Model parameter updates are converted to 8-bit fixed point values by applying 8-bit probabilistic quantization with 7 fractional bits [11]. The results show that in all three datasets, with a dropout rate $\delta = 0.3$, **Eagle** always outperforms previous works in terms of total computation.

# 9 CONCLUSION

We have studied the problem of stragglers in FL, which have a non-negligible impact on the performance and robustness of SA protocols. To cope with this problem, we have considered stragglers as client dropouts and developed two new SA protocols, namely **Eagle** and **Owl**. **Eagle** in SyncFL does not depend on dropouts anymore, and hence is more efficient than existing works, especially when the number stragglers is non-negligible. **Owl** in AsyncFL does not suffer from stragglers inherently, and is thus more efficient than the only existing solution in asynchronous settings.

As part of future work, we aim to optimize the cost of **Owl** and to consider stronger threat models whereby both FL clients and the server can be malicious and modify the actual aggregate model. In such a setting, honest FL clients should be able to verify the correctness of the computation of the aggregate value.

# ACKNOWLEDGMENTS

We thank the ARES reviewers for their comments. We also thank Mohamed Mansouri for his help with **FTSA**, Yiping Ma for his assistance with **Flamingo**, and Lucia Innocenti for the helpful discussions about client selection and dropouts. This work has been supported by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002, by the TRAIN project ANR-22-FAI1-0003-02, and by the ANR JCJC project Fed-BioMed 19-CE45-0006-01.

# REFERENCES


[1] 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*. 1253–1269.

[2] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *Proceedings of machine learning and systems* 1 (2019), 374–388.

[3] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security – CCS '17*. 1175–1191.

[4] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097* (2018).

[5] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthu Venkitasubramaniam, and Ruihan Wang. 2021. Diogenes: lightweight scalable RSA modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 590–607.

[6] Yann Fraboni, Richard Vidal, Laetitia Kameni, and Marco Lorenzi. 2021. Clustered sampling: Low-variance and improved representativity for clients selection in federated learning. In *Proceedings of the International Conference on Machine Learning – ICML '21*. 3407–3416.

[7] Yann Fraboni, Richard Vidal, Laetitia Kameni, and Marco Lorenzi. 2023. A General Theory for Federated Optimization with Asynchronous and Heterogeneous Clients Updates. *Journal of Machine Learning Research* 24, 110 (2023), 1–43.

[8] Marc Joye and Benoît Libert. 2013. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Proceedings of the International Conference on Financial Cryptography and Data Security – FC '13*. 111–125.

[9] Swanand Kadhe, Nived Rajaraman, O Ozan Koyluoglu, and Kannan Ramchandran. 2020. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *arXiv preprint arXiv:2009.11248* (2020).

[10] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning* 14, 1–2 (2021), 1–210.

[11] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[12] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* 55, 5 (2014).

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[14] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. 2023. LERNA: Secure Single-Server Aggregation via Key-Homomorphic Masking. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 302–334.

[15] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. 2023. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. *Cryptology ePrint Archive, Paper 2023/486* (2023).

[16] Mohamad Mansouri, Melek Önen, and Wafa Ben Jaballah. 2022. Learning from Failures: Secure and Fault-Tolerant Aggregation for Federated Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference – ACSAC '22*. 146–158.

[17] Mohamad Mansouri, Melek Önen, Wafa Ben Jaballah, and Mauro Conti. 2023. Sok: Secure aggregation based on cryptographic schemes for federated learning. *Proceedings on Privacy Enhancing Technologies – PoPETS '23* (2023), 140–157.

[18] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics – AISTATS '17*, Vol. 54. 1273–1282.

[19] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy – SP '19*. 739–753.

[20] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. 2022. Federated learning with buffered asynchronous aggregation. In *Proceedings of the International Conference on Artificial Intelligence and Statistics – AISTATS '22*. 3581–3607.

[21] Takashi Nishide and Kouichi Sakurai. 2011. Distributed Paillier Cryptosystem without Trusted Dealer. In *Proceedings of the International Workshop on Information Security Applications – WISA '11*. 44–60.

[22] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques – EUROCRYPT '99*. 223–238.

[23] Dario Pasquini, Danilo Francati, and Giuseppe Ateniese. 2022. Eluding secure aggregation in federated learning via model inconsistency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2429–2443.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[25] 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*. 89–104.

[26] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[27] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* (1979), 612–613.

[28] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy – SP '17*. 3–18.

[29] 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, 1 (2021), 479–489.

[30] Jinhyun So, Chaoyang He, Chien-Sheng Yang, Songze Li, Qian Yu, Ramy E Ali, Basak Guler, and Salman Avestimehr. 2022. Lightsecagg: a lightweight and versatile design for secure aggregation in federated learning. *Proceedings of Machine Learning and Systems* 4 (2022), 694–720.

[31] 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, Paper 2019/1136* (2019).

[32] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2019. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934* (2019).

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

[34] Qian Yu, Netanel Raviv, Jinhyun So, and Amir Salman Avestimehr. 2018. Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy. *CoRR* abs/1806.00939 (2018). arXiv:1806.00939

# A HYBRID SECURITY PROOFS

We present the security proofs of our two protocols using the hybrid argument technique. We do not consider the security proofs with an honest server since those proofs are relatively straightforward. We let the reader refer to [3] for more details.

We prove our protocols in the random oracle model for both adversarial models (i.e. honest-but-curious and active) since the security of the **JL** scheme and its **TJL** variant relies on the hash function being seen as a random oracle.

## A.1 Honest-but-Curious Model

We first prove our protocol **Eagle** secure in the honest-but-curious model and then our protocol **Owl**.

***Eagle***. We prove the security of **Eagle** against honest-but-curious server and clients through a sequence of hybrids. We define a simulated execution, represented by a simulator $\mathsf{SIM}_C^{\mathcal{U},t,\lambda}(\vec{x}_{\tau,C},\mathcal{U},\mathcal{U}_{on}^{(\tau)},\mathcal{U}_{shares}^{(\tau)})$ (denoted as $\mathsf{SIM}$ for short), through several modifications to the real execution of the protocol, represented by $\mathsf{REAL}_C^{\mathcal{U},t,\lambda}(\vec{x}_{\tau,\mathcal{U}},\mathcal{U},\mathcal{U}_{on}^{(\tau)},\mathcal{U}_{shares}^{(\tau)})$ (denoted as $\mathsf{REAL}$ for short), such that two subsequent hybrids are computationally indistinguishable.

$\mathtt{Hybrid}_0$. Let $C \subseteq \mathcal{U}$ be any subset of adversarial parties (server and clients) such that there are strictly less than $t$ adversarial clients. This hybrid is distributed exactly as $\mathsf{REAL}$, that is the joint view of the parties in $C$ in a real execution.

$\mathtt{Hybrid}_1$. In this hybrid, the behavior of the honest parties in $\mathcal{U} \setminus C$ is changed. Instead of using the key $c_{u,v} \leftarrow$ **KA.agree**$(pp^{KA}, c_u^{SK}, c_v^{PK})$ to encrypt messages, a random key $c_{u,v}$ is uniformly chosen by $\mathsf{SIM}$. The security of the Diffie-Hellman key agreement **KA** under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from $\mathtt{Hybrid}_0$.

$\mathtt{Hybrid}_2$. In this hybrid, all ciphertexts encrypted by honest parties in $\mathcal{U}\setminus C$ are changed. Instead of encryptions of shares $[sk_u]_v$, encryptions of 0 padded to the appropriate length are sent to other honest parties. Note that the honest clients in $\mathcal{U}\setminus C$ still answer with the correct shares $[sk_u]_v$ in the *reconstruction step*. Only the contents of the ciphertexts have changed, hence the IND-CPA security of **AE** applies and guarantees that this hybrid is indistinguishable from $\mathtt{Hybrid}_1$.

$\mathtt{Hybrid}_3$. In this hybrid, all shares of $sk_u$ generated by users in $\mathcal{U}\setminus C$ are substituted with shares of 0 using a different sharing of 0 for every client in that set. The security of **TJL** has its secret sharing properties ensuring that the distribution of any $|C|$ shares of 0 is identical to the distribution of $|C|$ shares of $sk_u$. The server never receives sufficient shares to reconstruct $sk_u$ since honest parties will not send their shares of $sk_u$. Therefore, this hybrid is indistinguishable from $\mathtt{Hybrid}_2$.

$\mathtt{Hybrid}_4$. In this hybrid, instead of using $\vec{y}_{u,\tau} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau}, \tau_0, \vec{x}_{u,\tau})$ for parties in $\mathcal{U} \setminus C$, we use $\vec{y}_{u,\tau} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau}, \tau_0, \vec{w}_{u,\tau})$, where $\sum_{u\in\mathcal{U}\setminus C}\vec{x}_{u,\tau} = \sum_{u\in\mathcal{U}\setminus C}\vec{w}_{u,\tau}$ for uniformly random $\vec{w}_{u,\tau}$. The security of **JL** under the Decision Composite Residuosity assumption ensures that encryptions of private inputs $\vec{x}_{u,\tau}$ are indistinguishable from encryptions of $\vec{w}_{u,\tau}$. Hence, this hybrid is indistinguishable from $\mathtt{Hybrid}_3$.

$\mathtt{Hybrid}_5$. In this hybrid, for all parties in $\mathcal{U} \setminus C$, instead of encryptions of the **JL** secret key $sk_{u,\tau}$, encryptions of 0 padded to the appropriate length are generated. The security of **TJL** under the Decision Composite Residuosity assumption ensures that this hybrid is indistinguishable from $\mathtt{Hybrid}_4$.

$\mathtt{Hybrid}_6$. Let us define $\mathcal{U}^* = \mathcal{U} \setminus C$ when $|\mathcal{U}_{on}^{(\tau)}| < t$ and $\mathcal{U}^* = \mathcal{U} \setminus \mathcal{U}_{on}^{(\tau)} \setminus C$ otherwise. In this hybrid, for all parties in $\mathcal{U}^*$, instead of computing $[\langle sk'_{0,\tau}\rangle]_u \leftarrow$ **TJL.ShareProtect**$(pp, \{[sk_v]_u\}_{v\in\mathcal{U}_{on}^{(\tau)}}, \tau)$, we set it to be a uniformly random variable of the appropriate size. Note that the key $sk_u$ is chosen uniformly at random in the real execution and that in a previous hybrid, the shares $[sk_u]_v$ of $sk_u$ given to the adversary were substituted with shares of 0. Hence, the only change in this hybrid is the substitution of the output of the algorithm **TJL.ShareProtect**. From the security of **TJL** under the Decision Composite Residuosity assumption, this hybrid is indistinguishable from $\mathtt{Hybrid}_5$.

We finally define the PPT simulator $\mathsf{SIM}$ that samples from the distribution described in $\mathtt{Hybrid}_6$. This proves that $\mathsf{SIM}$'s output is computationally indistinguishable from $\mathsf{REAL}$'s output.

***Owl***. We prove the security of **Owl** against honest-but-curious server and clients through a sequence of hybrids. We define a simulated execution, represented by a simulator $\mathsf{SIM}_C^{\mathcal{U},t,\lambda}(\vec{x}_C,\mathcal{U},\mathcal{U}_{on},\mathcal{U}_{shares})$ (denoted as $\mathsf{SIM}$ for short), through several modifications to the real execution of the protocol, represented by $\mathsf{REAL}_C^{\mathcal{U},t,\lambda}(\vec{x}_{\mathcal{U}},\mathcal{U},\mathcal{U}_{on},\mathcal{U}_{shares})$ (denoted as $\mathsf{REAL}$ for short), such that two subsequent hybrids are computationally indistinguishable.

$\mathtt{Hybrid}_0$. Let $C \subseteq \mathcal{U}$ be any subset of adversarial parties (server and clients) such that there are strictly less than $t$ adversarial clients. This hybrid is distributed exactly as $\mathsf{REAL}$, that is the joint view of the parties in $C$ in a real execution.

$\mathtt{Hybrid}_1$. In this hybrid, the behavior of the honest parties in $\mathcal{U} \setminus C$ is changed. Instead of using the key $c_{u,v} \leftarrow$ **KA.agree**$(pp^{KA}, c_u^{SK}, c_v^{PK})$ to encrypt messages, a random key $c_{u,v}$ is uniformly randomly chosen by $\mathsf{SIM}$. The security of the Diffie-Hellman key agreement **KA** under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from $\mathtt{Hybrid}_0$.

$\mathtt{Hybrid}_2$. In this hybrid, all ciphertexts encrypted by honest parties in $\mathcal{U} \setminus C$ are changed. Instead of encryptions of shares $[sk_{u,\tau_u}]_v$, encryptions of 0 padded to the appropriate length are sent to other honest parties. Note that the honest clients in $\mathcal{U} \setminus C$ still answer with the correct shares $[sk_{u,\tau_u}]_v$ in the *reconstruction step*. Only the contents of the ciphertexts have changed, hence the IND-CPA security of **AE** applies and guarantees that this hybrid is indistinguishable from $\mathtt{Hybrid}_1$.

$\mathtt{Hybrid}_3$. In this hybrid, all shares of $sk_{u,\tau_u}$ generated by users in $\mathcal{U} \setminus C$ are substituted with shares of 0 using a different sharing of 0 for every client in that set. The properties of **SS** guarantees that the distribution of any $|C|$ shares of 0 is identical to the distribution of $|C|$ shares of $sk_u$. The server never receives sufficient shares to reconstruct $sk_{u,\tau_u}$ since honest parties will not send their shares of $sk_{u,\tau_u}$. Therefore, this hybrid is indistinguishable from $\mathtt{Hybrid}_2$.

$\mathtt{Hybrid}_4$. In this hybrid, instead of using $\vec{y}_{u,\tau_u} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau_u}, \tau_0, \vec{x}_{u,\tau_u})$ for parties in $\mathcal{U} \setminus C$, we use $\vec{y}_{u,\tau_u} \leftarrow$ **JL.Protect**$(pp, sk_{u,\tau_u}, \tau_0, \vec{w}_{u,\tau_u})$, where $\sum_{u\in\mathcal{U}\setminus C} \vec{x}_{u,\tau_u} = \sum_{u\in\mathcal{U}\setminus C} \vec{w}_{u,\tau_u}$ for uniformly random $\vec{w}_{u,\tau_u}$. The security of **JL** under the Decision Composite Residuosity assumption ensures that encryptions of private inputs $\vec{x}_{u,\tau_u}$ are indistinguishable from encryptions of $\vec{w}_{u,\tau_u}$. Hence, this hybrid is indistinguishable from $\mathtt{Hybrid}_3$.

We finally define the PPT simulator SIM that samples from the distribution described in $\mathtt{Hybrid}_4$. This proves that SIM's output is computationally indistinguishable from REAL's output.

## A.2 Active Model

Here, we consider active adversaries, that are clients or server deviating from the protocol by sending incorrect and/or arbitrarily chosen messages to honest users, aborting, omitting messages and sharing their protocol view with each other (including the server), as defined in [3].

We assume that authenticated encrypted channels exist to ensure that received messages come from clients and not the server. We thus prevent the server from launching Sybil attacks. We also ask the server to forward clients public keys in an honest way during the *registration step*, to assist clients to establish private and authenticated communication channels with each other.

Similarly to [3], we include a consistency check phase to prevent the server to give different information about which client is online during the online phase. This allows us to avoid the server to learn different sets of shares from different users, allowing the unauthorised reconstruction of secrets.

Finally, the security proof requires us to be in the random oracle model. Such a model helps the simulator reprogram the random oracle to make dummy information indistinguishable from values of honest clients.

We only present the security proof in the active model for **Eagle** since the differences between the latter and the protocol **Owl** are small. Informally, the security of the protocol **Owl** partly relies on the security of **JL** and of **SS**, rather than of **TJL** with **ISS**. The remaining for proving our AsyncFL protocol **Owl** secure follows the same pattern as for **Eagle** (i.e. encryption scheme, signature scheme).

**Eagle** - We prove the security of **Eagle** against active server and clients through a sequence of hybrids. Given $n, t, \lambda$ and a subset $C$ of adversarial parties, we define $M_C$ as a probabilistic polynomial-time algorithm that denotes the "next-message" function of adversarial parties [3]. This function allows parties in $C$ to dynamically select their inputs at any round of the protocol execution as well as the list of online users.

We define a simulated execution, represented by a simulator $\mathtt{SIM}_C^{\mathcal{U},t,\lambda,\mathtt{ID}^{\xi}}(M_C)$ (denoted as SIM for short), through several modifications to the real execution of the protocol, represented by $\mathtt{REAL}_C^{\mathcal{U},t,\lambda}(M_C, \vec{x}_{\mathcal{U}\setminus C})$ (denoted as REAL for short), such that two subsequent hybrids are computationally indistinguishable. REAL exhibits the combined views of the adversarial parties in the protocol execution such that their messages and honest clients' aborts are chosen using $M_C$.

$M_C$ enables to dynamically set the subset of honest clients for which the server learns their local model aggregation. Therefore, this aggregation cannot be provided for a fixed subset of clients as input to SIM. Instead, SIM will make a single query to an ideal functionality ID (seen as a random oracle) that allows it to learn the aggregation for a dynamically chosen subset $\mathcal{L}$ of honest clients, such that $|\mathcal{L}| \geq \xi$ for a lower bound $\xi$ of the number of honest clients. Note that all parties and $M_C$ have access to the random oracle.

$\mathtt{Hybrid}_0$. Let $C \subseteq \mathcal{U}$ be any subset of adversarial parties (server and clients) such that there are strictly less than $t$ adversarial clients. This hybrid is distributed exactly as the joint view of $M_C$ in REAL, that is the joint view of the parties in $C$ in a real execution.

$\mathtt{Hybrid}_1$. In this hybrid, a simulator emulates the real execution. This simulator knows all the inputs $\vec{x}_{u,\tau}$ of the honest parties and runs a full execution of the protocol with $M_C$, including a simulation of the random oracle "on the fly", the secure communication channel establishment and the setup phase. Thus, the adversarial view is the same as in $\mathtt{Hybrid}_0$.

$\mathtt{Hybrid}_2$. In this hybrid, given any pair of honest users $u$ and $v$, the messages between $u$ and $v$ are encrypted before being given to $M_C$ and decrypted after being given to $M_C$, using a uniformly random key rather than the one obtained from **KA.agree**$(pp^{KA}, c_u^{SK}, c_v^{PK})$. The security of the Diffie-Hellman key agreement **KA** under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from $\mathtt{Hybrid}_1$.

$\mathtt{Hybrid}_3$. In this hybrid, SIM aborts if $M_C$ manages to deliver a message to an honest client $u$ on behalf of another honest client $v$ during the key setup phase, such that the message is different from the message that SIM has given to $M_C$ in that phase and that the decryption of this message does not fail (using the proper key). Note that the encryption key that $u$ and $v$ used in $\mathtt{Hybrid}_1$ was randomly chosen. Hence, based on such a message, the integrity of the ciphertext could be threatened. Since the underlying encryption scheme is INT-CTXT secure, then this hybrid is indistinguishable from the previous one.

$\mathtt{Hybrid}_4$. In this hybrid, the simulator changes all encrypted shares sent between pairs of honest users with encryptions of 0. Note that SIM still returns the "real" shares in the *reconstruction step* as it did before. Since the encryption keys were chosen uniformly at random, the IND-CPA security of the underlying encryption scheme guarantees that $\mathtt{Hybrid}_4$ is indistinguishable from $\mathtt{Hybrid}_3$.

$\mathtt{Hybrid}_5$. In this hybrid, SIM aborts if $M_C$ gives a signature on a set which correctly verifies based on an honest client's public key during the *consistency check step*, but the honest client has never created the signature on that set. The security of the underlying signature scheme guarantees that forgeries happen with negligible probability. Hence, $\texttt{Hybrid}_5$ is indistinguishable from $\texttt{Hybrid}_4$.

$\texttt{Hybrid}_6$. Let $Q \subseteq \mathcal{U}$ be the single set where an honest party received $Q$ during the *consistency check step* and then received at least $t$ valid signatures on it during the *reconstruction step*. SIM aborts if $M_C$ asks for key shares for some honest client $u$ either before the adversary has received the responses from the honest parties in the *reconstruction step*, or after such responses have been received for $u \notin Q$. In both cases, the key $sk_u$ is information theoretically hidden from $M_C$, and the simulator aborts if $M_C$ can guess one of those $sk_u$, happening with negligible probability. Indeed, the values $sk_u$ are chosen from the large domain $\mathbb{Z}_{N_0^2}$. Thus, the adversarial view is the same as in $\texttt{Hybrid}_5$.

$\texttt{Hybrid}_7$. In this hybrid, the simulator aborts if $M_C$ asks for encryptions of the per-round **JL** secret key for some honest client $u$ either before the adversary has received the responses from the honest parties in the *reconstruction step*, or after such responses have been received for $u \notin Q$. In both cases, the key $sk_{u,\tau}$ is information theoretically hidden from $M_C$, and the simulator aborts if $M_C$ can guess one of those $sk_{u,\tau}$, happening with negligible probability. Indeed, the values $sk_{u,\tau}$ are chosen from the large domain $\mathbb{Z}_{N_1^2}$. Thus, the adversarial view is the same as in $\texttt{Hybrid}_6$.

$\texttt{Hybrid}_8$. In this hybrid, the values $\vec{y}_{u,\tau}$ computed by SIM on behalf of honest users and sent to $M_C$ during the protection phase are changed with uniformly sampled values such that those values are independent from the rest of the view. Hence, this hybrid is indistinguishable from the previous one.

$\texttt{Hybrid}_9$. For all $u \in Q \setminus C$, we choose values $\vec{w}_{u,\tau}$ such that $\sum_{u \in Q \setminus C} \vec{w}_{u,\tau} = \sum_{u \in Q \setminus C} \vec{x}_{u,\tau}$. Since $sk_{u,\tau}$ is never queried for $u \in Q \setminus C$ by $M_C$, then in the view of $M_C$, the above values are identically distributed as in the previous hybrid.

$\texttt{Hybrid}_{10}$. In this hybrid, SIM does not receive the inputs of honest parties. Instead, during the *reconstruction step*, the simulator submits a query to ID for the set $Q \setminus C$ and uses the output to sample the required elements $\vec{w}_{u,\tau}$. By construction, we have $|Q| \geq t$ and $|Q \setminus C| \geq t - n_C$ where $n_C = \mathcal{U} \cap C$. Hence, ID will not abort. This change does not modify the view of the adversary, making $\texttt{Hybrid}_9$ and $\texttt{Hybrid}_{10}$ indistinguishable. Moreover, this hybrid does not make use of the inputs of honest parties, concluding the proof.