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

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

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

Marco Lorenzi  
Inria / Université Côte d’Azur  
Sophia Antipolis, France  
marco.orenzi@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 $\times4$; (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

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...
In the context of SyncFL, we significantly reduce the computation 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 ($|\mathcal{U}| = n_{\text{tot}}$) collaboratively trains a global model $\bar{x} \in \mathbb{R}^d$ 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 ($|\mathcal{U}^{(\tau)}| = n \leq n_{\text{tot}}$) through client selection [6, 7, 18]. Each client $u \in \mathcal{U}^{(\tau)}$ trains the model $\tilde{x}_u^{\tau}$ 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 $\tilde{x}_u^{\tau+1}$ 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: $\tilde{x}_r^{\tau+1} \leftarrow \frac{1}{n} \sum_{u \in \mathcal{U}^{(\tau)}} \tilde{x}_u^{\tau+1}$. This iterative process proceeds until the global model $\bar{x}$ 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, $n_{\text{tot}} \in [10^5, 10^{10}]$ and $n \in [50, 5000]$ [10].

Asynchronous Federated Learning. Asynchronous FL (AsyncFL) allows clients not to synchronize 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 buffer for later use. If 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.

<table>
<thead>
<tr>
<th>Client Comp.</th>
<th>Client Comm.</th>
<th>Online Rounds</th>
<th>FL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecAgg [3]</td>
<td>$O(n^2 + nd)$</td>
<td>$O(n + d)$</td>
<td>4</td>
</tr>
<tr>
<td>FTSA [16]</td>
<td>$O(n^2 + n \log(n)d)$</td>
<td>$O(n + d)$</td>
<td>3</td>
</tr>
<tr>
<td>Eagle</td>
<td>$O(n \log(n) + d)$</td>
<td>$O(n + d)$</td>
<td>3</td>
</tr>
<tr>
<td>LightSecAgg [30]</td>
<td>$O(n^2 + \frac{d}{(1-\delta)n-1} + d)$</td>
<td>$O(n \frac{d}{(1-\delta)n-1} + d)$</td>
<td>3</td>
</tr>
<tr>
<td>Owl</td>
<td>$O(n^2 + d)$</td>
<td>$O(n + d)$</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Complexity analysis for one round ($n$: number of clients; $\delta$: fraction of dropped clients; $\tau$: threshold value; $d$: input dimension).
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 $\tilde{x}_u, \tau_u$. 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 AsyecFL.

**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 $D_u$ private during training, adversaries who have access to the clients’ updated model parameters can infer information about $D_u$ [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:

- $(s_{k_0}, \{s_{k_u}\}_{u \in [1, n]}, N, H) \leftarrow$ **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 $s_{k_u} \leftarrow \mathbb{Z}_{N^2}$ and sets the aggregator key $s_{k_0} = -\sum_{u=1}^{n} s_{k_u}$. 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**($pp, s_{k_u}, \tau, x_{u, \tau}$): This algorithm encrypts the private input $x_{u, \tau} \in \mathbb{Z}_N$ for time period $\tau$ using secret key $s_{k_u} \in \mathbb{Z}_{N^2}$. It outputs the cipher $y_{u, \tau} = (1 + x_{u, \tau} N) \cdot H(\tau)^{s_{k_u}} \text{ mod } N^2$.

- $x_{\tau} \leftarrow$ **JL.Aggregate**($pp, s_{k_0}, \tau, \{y_{u, \tau}\}_{u \in [1, n]}$): This algorithm aggregates the $n$ ciphers received at time period $\tau$ to obtain $y_{\tau} = \prod_{u=1}^{n} y_{u, \tau}$ and decrypts the result $x_{\tau} = \sum_{u=1}^{n} x_{u, \tau} = \frac{H(\tau)^{s_{k_0}} \cdot q_{\tau} - 1}{N} \text{ mod } N$.

The JL scheme ensures Aggregator Obliviousness 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 $s_k(u)$ is defined in $\mathbb{Z}_p^*$ (if $N$ is prime) and is not defined over integers rather than in a field (see Section 3.2 [16]). Informally, the design principles are the following: (1) usually assumed to be numerous), only one key is periodically

synchronous setting. The design principles are the following: (1)

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

Setup - Key Setup:

Client $u \in \mathcal{U}(t)$, // Generate TJS key and secret share
(1) $\sigma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $c_{\sigma,u} \leftarrow KA.agree(ppKA, c_{PK}^u, \sigma)$, // Establish pairwise

channel keys with each client
(2) $s_{\sigma,u} \leftarrow \mathbb{Z}_p^*$; // Generate TJS secret key
(3) $\{ \langle v \rightarrow \mathcal{U}(t) \setminus \{u\} \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$. // Generate t-out-of-n shares

(4) $\gamma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $\epsilon_{\gamma,u} \leftarrow AE.enc(c_{\sigma,u}, 0 \mid \mathbb{Z}_p^*$), // Encrypt each share with the corresponding public key

(5) Send $\{ \langle v \leftarrow \mathcal{U}(t) \setminus \{u\} \leftarrow TJS.Recon(v, \mathcal{U}(t))$. // Recover the secret

Client $u \in \mathcal{U}(t)$, // Decrypt the received shares
(1) $\sigma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $c_{\sigma,u} \leftarrow AE.dec(c_{\sigma,u}, 0 \mid \mathbb{Z}_p^*$). // Decrypt each share with the corresponding public key

Online - Protection (step t):

Client $u \in \mathcal{U}(t)$, // Protect the private input using TJS, the per-round TJS secret key using TJS, and send them to the server
(1) $s_{\sigma,u} \leftarrow \mathbb{Z}_p^*$; // Generate the per-round TJS secret key

(2) $\tilde{s}_{u} \leftarrow TJS.Protect(pp, s_{\sigma,u}, \mathcal{U}(t))$. // Protect private input $\tilde{s}_{u,t} \in \mathbb{Z}_p^*$ using TJS

(3) $s_{\sigma,u} \leftarrow TJS.Protect(pp, s_{\sigma,u}, \mathcal{U}(t))$. // Protect the per-round TJS secret key $s_{\sigma,u} \in \mathbb{Z}_p^*$ using TJS

(4) Send $\tilde{s}_{u}$ and $(s_{\sigma,u})$ to Server.

Server: // If the number of online clients $(\mathcal{U}(t) \subseteq \mathcal{U}(t))$ is less than t, abort; otherwise, collect the secret key shares, the secret inputs, and broadcast $\mathcal{U}(t)$ to all clients

(1) Collect $\{ (s_{\sigma,u}) \}_{u \in \mathcal{U}(t)}$ and $(\tilde{s}_{u})_{u \in \mathcal{U}(t)}$.

(2) If $\mathcal{U}(t) < t$, abort; otherwise broadcast $\mathcal{U}(t)$.

Online - Consistency Check (step t+1): // See Figure 4 (Consistency Check) of [3]

Online - Reconstruction (step t+1):

Client $u \in \mathcal{U}(t)$, // Compute the share of the per-round TJS server key, and send it to the server
(1) $(s_{\sigma,u}) \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$.

(2) Send $(s_{\sigma,u})$ to Server.

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

(1) $(s_{\sigma,u})_{u \in \mathcal{U}(t)}$.

(2) If $\mathcal{U}(t) < t$, abort; otherwise, proceed.

(3) $s_{\sigma,u} \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$. // Reconstruct the zero-scalar value of the online clients

(4) $s_{\sigma,u} \leftarrow TJS.Aggregate(pp, 0, \tau, (s_{\sigma,u})_{u \in \mathcal{U}(t)}$, $t)$. // Complete the aggregation

Figure 1: Eagle in SyncFL.

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 $\mathcal{U}(t)$, such that $|\mathcal{U}(t)| \geq n$

Public Parameters: Generate the public parameters

$ppKA \leftarrow KA.Param(n)$, (1, $\langle u \rightarrow \mathcal{U}(t) \setminus \{u\} \leftarrow TJS.Setup(n)$ and $1L.Setup(n)$ s.t. $N_0 \geq 2 \cdot N_1 + \log_2(n)$ and set

$pp = (ppKA, N_0, N_1, H, H_0, H_1, \tau, \sigma, t, n, R)$

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

Setup - Key Setup:

Client $u \in \mathcal{U}(t)$, // Generate TJS key and secret share
(1) $\sigma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $c_{\sigma,u} \leftarrow KA.agree(ppKA, c_{PK}^u, \sigma)$, // Establish pairwise

channel keys with each client
(2) $s_{\sigma,u} \leftarrow \mathbb{Z}_p^*$; // Generate TJS secret key
(3) $\{ \langle v \rightarrow \mathcal{U}(t) \setminus \{u\} \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$. // Generate t-out-of-n shares

(4) $\gamma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $\epsilon_{\gamma,u} \leftarrow AE.enc(c_{\sigma,u}, 0 \mid \mathbb{Z}_p^*$), // Encrypt each share with the corresponding public key

(5) Send $\{ \langle v \leftarrow \mathcal{U}(t) \setminus \{u\} \leftarrow TJS.Recon(v, \mathcal{U}(t))$. // Recover the secret

Client $u \in \mathcal{U}(t)$, // Decrypt the received shares
(1) $\sigma \leftarrow \mathcal{U}(t) \setminus \{u\}$, $c_{\sigma,u} \leftarrow AE.dec(c_{\sigma,u}, 0 \mid \mathbb{Z}_p^*$). // Decrypt each share with the corresponding public key

Online - Protection (step t):

Client $u \in \mathcal{U}(t)$, // Protect the private input using TJS, the per-round TJS secret key using TJS, and send them to the server
(1) $s_{\sigma,u} \leftarrow \mathbb{Z}_p^*$; // Generate the per-round TJS secret key

(2) $\tilde{s}_{u} \leftarrow TJS.Protect(pp, s_{\sigma,u}, \mathcal{U}(t))$. // Protect private input $\tilde{s}_{u,t} \in \mathbb{Z}_p^*$ using TJS

(3) $s_{\sigma,u} \leftarrow TJS.Protect(pp, s_{\sigma,u}, \mathcal{U}(t))$. // Protect the per-round TJS secret key $s_{\sigma,u} \in \mathbb{Z}_p^*$ using TJS

(4) Send $\tilde{s}_{u}$ and $(s_{\sigma,u})$ to Server.

Server: // If the number of online clients $(\mathcal{U}(t) \subseteq \mathcal{U}(t))$ is less than t, abort; otherwise, collect the secret key shares, the secret inputs, and broadcast $\mathcal{U}(t)$ to all clients

(1) Collect $\{ (s_{\sigma,u}) \}_{u \in \mathcal{U}(t)}$ and $(\tilde{s}_{u})_{u \in \mathcal{U}(t)}$.

(2) If $\mathcal{U}(t) < t$, abort; otherwise broadcast $\mathcal{U}(t)$.

Online - Consistency Check (step t+1): // See Figure 4 (Consistency Check) of [3]

Online - Reconstruction (step t+1):

Client $u \in \mathcal{U}(t)$, // Compute the share of the per-round TJS server key, and send it to the server
(1) $(s_{\sigma,u}) \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$.

(2) Send $(s_{\sigma,u})$ to Server.

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

(1) $(s_{\sigma,u})_{u \in \mathcal{U}(t)}$.

(2) If $\mathcal{U}(t) < t$, abort; otherwise, proceed.

(3) $s_{\sigma,u} \leftarrow TJS.Share(c_{\sigma,u}, \mathcal{U}(t))$. // Reconstruct the zero-scalar value of the online clients

(4) $s_{\sigma,u} \leftarrow TJS.Aggregate(pp, 0, \tau, (s_{\sigma,u})_{u \in \mathcal{U}(t)}$, $t)$. // 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\(^\text{1}\). At the Key Setup step, each client independently computes \( t \) out of \( n \) shares of its secret key \( s_{uk} \) using \( \text{TJL.SEShare} \). 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 \( s_{uk,\tau} \) that is then used to protect their private input vectors \( x_{u,\tau} \) using \( \text{JL.Protect} \) at random \( \tau \), such that the protection with JL uses a fixed \( \tau = n \). This key is further protected using \( \text{TJL.Protect} \) 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 \( \mathcal{U}_\text{on}(\tau) \). Their goal is to help compute/reconstruct the per-aggregation round 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 \( \text{TJL.ShareCombine} \). Then, the actual model parameters \( \hat{x}_\tau \) can be computed using \( \text{JL.Aggregate} \).

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 corruptions) up to \( n - t \) clients. Let \( \mathcal{U}_{\text{corr}} \) be the set of corrupted clients and \( C = \mathcal{U}_{\text{corr}} \cup S \) where \( S \) represents the server. The view of the server is computationally indistinguishable from a simulated view if the number of corrupted clients is less than the threshold \( t \) (i.e., \( |\mathcal{U}_{\text{corr}}| < t \)). 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 \( s_{uk,\tau} \) from random values. It also ensures that if parties in \( C \) have access to at most \( t - 1 \) shares of \( s_{uk} \) (i.e., \( |\mathcal{U}_{\text{corr}}| < t \)), then, they cannot distinguish the shares held by the honest clients 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 \( |\mathcal{U}_\text{on}(\tau)| \geq |\mathcal{U}_\text{shares}(\tau)| \geq t \) and hence AO is ensured.

- In the active model, \( S \) can additionally manipulate its inputs to the protocol. The only messages \( S \) distributes, other than the clients’ public keys, are the protected shares that are forwarded from and to the clients. \( S \) cannot modify the values of these encrypted shares thanks to the underlying authenticated encryption scheme AE. Therefore, \( S \)’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 \( \mathcal{U}_\text{on}(\tau) \). It is important to note that \( S \) can present different views to different clients regarding their online/dropped status. This capability enables \( S \) to easily acquire the individual temporary JL key \( s_{uk,\tau} \) of a client \( u \). More precisely, \( S \) can convince a subset of honest clients that

\( ^{\text{1}} \) 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].
the set of online clients is $U^{(t)}_{on}$ while indicating to another subset that the online clients’ set is $U^{(t)}_{on} = U^{(t)}_{on} \setminus \{u\}$ (i.e., $u$ is dropped). If this occurs, $S$ can aggregate the protected inputs from $U^{(t)}_{on}$ to derive the per-round aggregation key $sk_r$, and also aggregate inputs from $U^{(t)}_{on}$ to derive $sk'_r$, and then calculate $sk_{u,r} = sk_r - sk'_r$. Considering the scenario where $S$ may collude with $n-t$ corrupted clients, it can obtain $n-t$ shares of $(sk_r)$ and $(sk'_r)$ and hence $n-t$ shares of $sk_{u,r}$. Furthermore, $S$ has the ability to convince $\frac{1}{2}$ honest clients that $u$ is online, and the other $\frac{1}{2}$ honest clients that $u$ is dropped, thereby collecting shares of $\tilde{y}_r$ and $\tilde{y}'_r$ respectively. Hence, in total, the server can learn a maximum number of $n-t+\frac{1}{2}$ shares of $(sk_r)$ and $(sk'_r)$. Therefore, to prevent such attacks and still ensure $AO$, we additionally require that $n-t+\frac{1}{2} < t \implies t > \frac{2n}{3}$. Hence the protocol can recover from up to $\frac{2}{3} - 1$ client failures in the active model.

To conclude, if the server operates under an honest-but-curious model, selecting $t > \frac{2}{3}$ 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 $r$. 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 $sk_{u,\tau_0}$. 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.Aggregates 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 $\tau_0$, denoted as $sk_{u,\tau_0}$. 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 $\tilde{x}_{u,\tau_0}$ (vector) is protected using JL.Protect which takes as inputs $sk_{u,\tau_0}$ 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 $\sum_{v \in U_{on}}[sk_{u,\tau_0}]_v$. 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.Aggr.

5.2 Security Analysis (Sketch)

We briefly analyze 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 $\tilde{y}_r$ 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 $sk_{u,\tau_0}$ (i.e. $|U_{corr}| < t$), 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 $|\{U^{(t)}_{on}\}| \geq |\{U^{(t)}_{shares}\}|$.

- When $S$ is an active adversary, it can try to convince a subset of honest clients that the set of online clients is $U^{(t)}_{on}$ while indicating to another subset that the set of online clients is $U^{(t)}_{on} = U^{(t)}_{on} \setminus \{u\}$ for a client $u$. If this occurs, $S$ can reconstruct $sk_r$ and $sk'_r$ and then, it can compute $sk_{u,\tau_0} = sk_r - sk'_r$. Because we assume that there are $n-t$ corrupted clients, $S$ can obtain $n-t$ shares of $sk_0$ and $sk'_0$ respectively. Furthermore, $S$ has the ability to convince $\frac{1}{2}$ honest clients that the client $u$ is online, and the other $\frac{1}{2}$ honest clients that $u$ is dropped, thereby collecting shares of $sk_0$ and $sk'_0$ respectively. Therefore, to ensure $AO$, we require that $n-t+\frac{1}{2} < t \implies t > \frac{2n}{3}$. 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 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 $x_u$ 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 $b_u$ 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 $sk'_{u,t}$, 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 $sk'_{u,t}$ 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 unmask the result, which requires a computation cost of $O(\delta n d)$. 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.

<table>
<thead>
<tr>
<th></th>
<th>Flamingo [15]</th>
<th>Eagle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client Comp.</strong></td>
<td>Regular client: $O(k^2 + ad)$</td>
<td>Regular client: $O(d)$</td>
</tr>
<tr>
<td></td>
<td>Decryptor: $O(\delta an + (1 - \delta)n)$</td>
<td>Decryptor: $O(1 - \delta)n + k \log k$</td>
</tr>
<tr>
<td><strong>Client Comm.</strong></td>
<td>Regular client: $O(\delta an + (1 - \delta)n)$</td>
<td>Regular client: $O(d)$</td>
</tr>
<tr>
<td></td>
<td>Decryptor: $O(\delta an + (1 - \delta)n)$</td>
<td>Decryptor: $O(n)$</td>
</tr>
</tbody>
</table>

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-scaler value becomes $O(k \log k)$, resulting in a total cost of $O(n^2 + k \log k)$. 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 ciphertext 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 unmask 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 $O((1 - \delta)n k)$. The reconstruction of the pairwise masks implies a cost of $O(\delta nk)$, and then the aggregation and unmasking cost $O(k^2 + \delta nk + (1 - \delta)n k + nd)$.

  - 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 $x_{u,\tau_u}$ 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 $O(n^2 + \delta nk + (1 - \delta)n k + d)$, 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 $y_{u,\tau_u}$ 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 $O(n^2 + \delta nk + (1 - \delta)n k + d)$ 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 unmask 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
8.1 Experimental Setting

All our implementations use the Python programming language\(^\dagger\). Experiments were carried out on a single-threaded process, using a machine equipped with an Intel(R) Core(TM) i7-7800X CPU @ 3.5GHz 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 \( t = \frac{\delta}{2} \). We assume that client dropouts happen before the clients send their protected inputs. This is essentially the "worst

\[\text{use cases of training a machine learning model for MNIST, CIFAR-10 and Shakespeare datasets.}\]

\[\text{\textdagger} The \text{code of the paper can be found at: GitHub repository.}\]

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\(^\dagger\). Experiments were carried out on a single-threaded process, using a machine equipped with an Intel(R) Core(TM) i7-7800X CPU @ 3.5GHz 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 \( t = \frac{\delta}{2} \). We assume that client dropouts happen before the clients send their protected inputs. This is essentially the "worst

\[\text{\textdagger} The code of the paper can be found at: GitHub repository.\]

use cases of training a machine learning model for MNIST, CIFAR-10 and Shakespeare datasets.
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 δ = 0.1, Eagle is approximately 10X 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 4k. 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 (δ ≥ 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 δ = 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.](image)

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.
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


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 $\text{SIM}_C(U, C, U_{on}, U_{shares})$ (denoted as SIM for short), through several modifications to the real execution of the protocol, represented by $\text{REAL}_C(U, C, U_{on}, U_{shares})$ (denoted as REAL for short), such that two subsequent hybrids are computationally indistinguishable.

**Hybrid 0.** Let $C \subseteq U$ 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 REAL, that is the joint view of the parties in $U \setminus C$ in a real execution.

**Hybrid 1.** In this hybrid, the behavior of the honest parties in $U \setminus C$ is changed. Instead of using the key $c_{u,v} \leftarrow \text{KA.agree}(p p, c_{u,v}, c_{v})$ to encrypt messages, a random key $c_{u,v}$ is uniformly chosen by SIM. The security of the Diffie-Hellman key agreement KA under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from Hybrid 0.

**Hybrid 2.** In this hybrid, all ciphertexts encrypted by honest parties in $U \setminus C$ are changed. Instead of encryptions of shares $[s k_u]$ of $s k_u$ to $w_u, r$ padded to the appropriate length are sent to other honest parties. Note that the honest clients in $U \setminus C$ still answer with the correct shares $[s k_u]$ 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 Hybrid 1.

**Hybrid 3.** In this hybrid, all shares of $s k_u$ generated by users in $U \setminus C$ 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 $s k_u$. The server never receives sufficient shares to reconstruct $s k_u$ since honest parties will not send their shares of $s k_u$. Therefore, this hybrid is indistinguishable from Hybrid 3.

**Hybrid 4.** In this hybrid, instead of using $w_u, t \leftarrow \text{JL.Protect}(p p, s k_u, r_0, x_u, r)$ for parties in $U \setminus C$, we use $w_u, t \leftarrow \text{JL.Protect}(p p, s k_u, r_0, \tilde{w}_u, r)$, where $\sum_{v \in U \setminus C} x_u, r = \sum_{v \in U \setminus C} \tilde{w}_u, r$ for uniformly random $\tilde{w}_u, r$. The security of JL under the Decisional Composite Residuosity assumption ensures that encryptions of private inputs $x_u, r$ are indistinguishable from encryptions of $\tilde{w}_u, r$. Hence, this hybrid is indistinguishable from Hybrid 4.

**Hybrid 4’.** Let us define $U^* = U \setminus C$ when $|U_{on}| < t$ and $U^* = U \setminus U_{on} \setminus C$ otherwise. In this hybrid, for all parties in $U^*$, instead of computing $([s k_u])_u \leftarrow \text{TJL.ShareProtect}(p p, [s k_u], u \in U_{on}, t, \lambda)$, we set it to be a uniformly random variable of the appropriate size. Note that the key $s k_u$ is chosen uniformly at random in the real execution and that in a previous hybrid, the shares $[s k_u]_o$ of $s k_u$ 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 Decisional Composite Residuosity assumption, this hybrid is indistinguishable from Hybrid 4. We define the PPT simulator SIM that samples from the distribution described in Hybrid 4. This proves that SIM’s output is computationally indistinguishable from 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 $\text{SIM}_C(U, C, U_{on}, U_{shares})$ (denoted as SIM for short), through several modifications to the real execution of the protocol, represented by $\text{REAL}_C(U, C, U_{on}, U_{shares})$ (denoted as REAL for short), such that two subsequent hybrids are computationally indistinguishable.

**Hybrid 0.** Let $C \subseteq U$ 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 REAL, that is the joint view of the parties in $U \setminus C$ in a real execution.

**Hybrid 1.** In this hybrid, the behavior of the honest parties in $U \setminus C$ is changed. Instead of using the key $c_{u,v} \leftarrow \text{KA.agree}(p K_v, c_{u,v})$ to encrypt messages, a random key $c_{u,v}$ is uniformly chosen by SIM. The security of the Diffie-Hellman key agreement KA under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from Hybrid 0.

**Hybrid 2.** In this hybrid, all ciphertexts encrypted by honest parties in $U \setminus C$ are changed. Instead of encryptions of shares $[s k_u]$ of $s k_u$ to $w_u, r$ padded to the appropriate length are sent to other honest parties. Note that the honest clients in $U \setminus C$ still answer with the correct shares $[s k_u]$ 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 Hybrid 1.

**Hybrid 3.** In this hybrid, all shares of $s k_u$ generated by users in $U \setminus C$ 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 $s k_u$. The server never receives sufficient shares to reconstruct $s k_u$ since honest parties will not send their shares of $s k_u$. Therefore, this hybrid is indistinguishable from Hybrid 2.

**Hybrid 4.** In this hybrid, instead of using $w_u, r \leftarrow \text{JL.Protect}(p p, s k_u, r_0, x_u, r)$ for parties in $U \setminus C$, we use $w_u, r \leftarrow \text{JL.Protect}(p p, s k_u, r_0, \tilde{w}_u, r)$, where $\sum_{v \in U \setminus C} x_u, r = \sum_{v \in U \setminus C} \tilde{w}_u, r$ for uniformly random $\tilde{w}_u, r$. The security of JL under the Decisional Composite Residuosity assumption ensures that encryptions of private inputs $x_u, r$ are indistinguishable from encryptions of $\tilde{w}_u, r$. Hence, this hybrid is indistinguishable from Hybrid 4.

**Hybrid 4’.** Let us define $U^* = U \setminus C$ when $|U_{on}| < t$ and $U^* = U \setminus U_{on} \setminus C$ otherwise. In this hybrid, for all parties in $U^*$, instead of computing $([s k_u])_u \leftarrow \text{TJL.ShareProtect}(p p, [s k_u], u \in U_{on}, t, \lambda)$, we set it to be a uniformly random variable of the appropriate size. Note that the key $s k_u$ is chosen uniformly at random in the real execution and that in a previous hybrid, the shares $[s k_u]_o$ of $s k_u$ 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 Decisional Composite Residuosity assumption, this hybrid is indistinguishable from Hybrid 4. We define the PPT simulator SIM that samples from the distribution described in Hybrid 4. This proves that SIM’s output is computationally indistinguishable from REAL’s output.
identical to the distribution of \(|C|\) shares of \(sk_u\). The server never receives sufficient shares to reconstruct \(sk_u, r_u\) since honest parties will not send their shares of \(sk_u, r_u\). Therefore, this hybrid is indistinguishable from \textbf{Hybr id}_2.

\textbf{Hybr id}_2. In this hybrid, instead of using \(\tilde{y}_{u, r_u} \leftarrow \textbf{JL.Protect}(pp, sk_{u, r_u}, r_0, \tilde{x}_{u, r_u})\) for parties in \(U \setminus C\), we use \(\hat{y}_{u, r_u} \leftarrow \textbf{JL.Protect}(pp, sk_{u, r_u}, r_0, \tilde{w}_{u, r_u})\), where \(\sum_{u \in U \setminus C} \tilde{x}_{u, r_u} = \sum_{u \in U \setminus C} \tilde{w}_{u, r_u}\) for uniformly random \(\tilde{w}_{u, r_u}\). The security of \textbf{JL} under the Decision Composite Residuosity assumption ensures that encryptions of private inputs \(\tilde{x}_{u, r_u}\) are indistinguishable from encryptions of \(\tilde{w}_{u, r_u}\). Hence, this hybrid is indistinguishable from \textbf{Hybr id}_3. We finally define the PPT simulator \textbf{SIM} that samples from the distribution described in \textbf{Hybr id}_4. This proves that \textbf{SIM}’s output is computationally indistinguishable from \textbf{REAL}’s output.

\subsection{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 client public keys in an honest way during the \textit{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 \textbf{Eagle} since the differences between the latter and the protocol \textbf{Owl} are small. Informally, the security of the protocol \textbf{Owl} partly relies on the security of \textbf{JL} and of \textbf{SS}, rather than of \textbf{TJL} with \textbf{ISS}. The remaining for proving our AsynceFL protocol \textbf{Owl} secure follows the same pattern as for \textbf{Eagle} (i.e. encryption scheme, signature scheme).

\textbf{Eagle} - We prove the security of \textbf{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 \(\text{SIM}^{U, \lambda, \lambda, DF}_C(M_C)\) (denoted as \(\text{SIM}\) for short), through several modifications to the real execution of the protocol, represented by \(\text{REAL}^{U, \lambda}_C(M_C, \tilde{x}_{U, C})\) (denoted as \(\text{REAL}\) for short), such that two subsequent hybrids are computationally indistinguishable. \textbf{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 \textbf{SIM}. Instead, \textbf{SIM} will make a single query to an ideal functionality \(\text{IO}\) (seen as a random oracle) that allows it to learn the aggregation for a dynamically chosen subset \(L\) of honest clients, such that \(|L| \geq \xi\) for a lower bound \(\xi\) of the number of honest clients. Note that all parties and \(M_C\) have access to the random oracle.

\textbf{Hybr id}_4. Let \(C \subseteq U\) 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 \textbf{REAL}, that is the joint view of the parties in \(C\) in a real execution.

\textbf{Hybr id}_5. In this hybrid, a simulator emulates the real execution. This simulator knows all the inputs \(\tilde{x}_{u, r_u}\) 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 \textbf{Hybr id}_4.

\textbf{Hybr id}_6. 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 \(\text{KA.agree}(pp^{KA, sk_u, c_{PK}})\). The security of the Diffie-Hellman key agreement \(KA\) under the Decisional Diffie-Hellman assumption ensures that this hybrid is indistinguishable from \textbf{Hybr id}_1.

\textbf{Hybr id}_7. In this hybrid, \textbf{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 \textbf{Hybr id}_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.

\textbf{Hybr id}_8. 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 \textit{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 \textbf{Hybr id}_8 is indistinguishable from \textbf{Hybr id}_7.

\textbf{Hybr id}_9. In this hybrid, \textbf{SIM} aborts if \(M_C\)’s signature on a set which correctly verifies based on an honest client’s public key during the \textit{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, Hybrid 5 is indistinguishable from Hybrid 4.

Hybrid 6. Let $Q \subseteq U$ 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 $s_{k_u}$ is information theoretically hidden from $M_C$, and the simulator aborts if $M_C$ can guess one of those $s_{k_u}$, happening with negligible probability. Indeed, the values $s_{k_u}$ are chosen from the large domain $Z_{N^2}$. Thus, the adversarial view is the same as in Hybrid 5.

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 $s_{k_{u,\tau}}$ is information theoretically hidden from $M_C$, and the simulator aborts if $M_C$ can guess one of those $s_{k_{u,\tau}}$, happening with negligible probability. Indeed, the values $s_{k_{u,\tau}}$ are chosen from the large domain $Z_{N^2}$. Thus, the adversarial view is the same as in Hybrid 6.

Hybrid 8. In this hybrid, the values $\tilde{w}_{u,\tau}$ 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.

Hybrid 9. For all $u \in Q \setminus C$, we choose values $\tilde{w}_{u,\tau}$ such that $\sum_{u \in Q \setminus C} \tilde{w}_{u,\tau} = \sum_{u \in Q \setminus C} \tilde{x}_{u,\tau}$. Since $s_{k_{u,\tau}}$ 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.

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 $\tilde{w}_{u,\tau}$. By construction, we have $|Q| \geq t$ and $|Q \setminus C| \geq t - n_C$ where $n_C = U \cap C$. Hence, ID will not abort. This change does not modify the view of the adversary, making Hybrid 9 and Hybrid 10 indistinguishable. Moreover, this hybrid does not make use of the inputs of honest parties, concluding the proof.