Per-element Secure Aggregation against Data Reconstruction Attacks in Federated Learning

Takumi Suimon*, Yuki Koizumi*, Junji Takemasa*, and Toru Hasegawa†
*Graduate School of Information Science and Technology, The University of Osaka
†Faculty of Materials for Energy, Shimane University

***Abstract*—Federated learning (FL) enables collaborative model training without sharing raw data, but individual model updates may still leak sensitive information. Secure aggregation (SecAgg) mitigates this risk by allowing the server to access only the sum of client updates, thereby concealing individual contributions. However, a significant vulnerability has recently attracted increasing attention: when model updates are sparse vectors, a non-zero value contributed by a single client at a given index can be directly revealed in the aggregate, enabling precise data reconstruction attacks. In this paper, we propose a novel enhancement to SecAgg that reveals aggregated values only at indices with at least t non-zero contributions. Our mechanism introduces a per-element masking strategy to prevent the exposure of under-contributed elements, while maintaining modularity and compatibility with many existing SecAgg implementations by relying solely on cryptographic primitives already employed in a typical setup. We integrate this mechanism into Flamingo, a low-round SecAgg protocol, to provide a robust defense against such attacks. Our analysis and experimental results indicate that the additional computational and communication overhead introduced by our mechanism remains within an acceptable range, supporting the practicality of our approach.**

***Index Terms*—Federated learning, Secure aggregation, Data reconstruction attacks**

I. Introduction

Federated learning (FL) [1] is widely adopted for its ability to train models on distributed data while preserving data privacy. In each training round, clients independently train the received global model on their datasets and send model updates to the server. Please note that models and updates are represented as vectors, and the terms "models/updates" and "vectors" are used interchangeably throughout this paper. The server aggregates these updates to produce a new global model, which is again distributed to clients. The privacy principle of FL is that only updates are shared under the assumption that these updates do not reveal the raw datasets. However, numerous studies [2], [3] have shown that a motivated server can often reconstruct clients' data from their individual updates.

Secure aggregation (SecAgg) [4]–[8] is a key countermeasure to this privacy risk. It ensures that the server receives only the aggregate of at least t vectors (where $t \geq 2$), without revealing any individual vector. When integrated into FL, SecAgg prevents the server from accessing individual updates and mitigates the risk of data reconstruction.

However, a critical vulnerability occurs when the vectors are sparse. SecAgg hides individual vectors but does not always hide individual elements. If only one client provides a non-zero value at a given index in the vector, that value can be revealed from the aggregate result. This loophole poses a serious privacy threat. Recent studies [9]–[14] propose data reconstruction attacks under SecAgg, where the server maliciously crafts global models so that updates from non-victim clients are zero at specific indices, exposing corresponding elements in the victim clients' updates.

To counter such attacks, two types of defenses have been proposed: model inconsistency checks and model integrity checks. However, both approaches rely on client-side model checks, which suffer from fundamental limitations. First, **model inconsistency checks** [12], [15] attempt to detect attacks by verifying whether all clients receive the same model, which prevents a malicious server from delivering different modified models to different clients. Unfortunately, this type of defense is not applicable to modern FL, such as personalized FL [16] and asynchronous FL [17], [18], where the server inherently distributes different models. In addition, model inconsistency checks fail to detect attacks where the same modified model is sent to all clients [10], [13], as no inconsistency arises to be detected.

Second, **model integrity checks** [19]–[23] aim to detect attacks by verifying whether the received model has been maliciously modified. This class of defenses falls into two main categories, each with drawbacks. One approach relies on cryptographic verification to ensure that the global model correctly reflects client updates [19]–[21]. While this provides strong security, it relies on costly cryptographic techniques such as zero-knowledge proofs, which significantly increase computational overhead. Another approach inspects the model for anomalous parameters or structures [22], [23]. While lightweight, in FL settings where clients receive only partial models [24], [25], it becomes impossible to verify the unseen parts. These parts are treated as zero elements, which opens the door to data reconstruction attacks.

This paper proposes a fundamental mechanism to prevent the exposure of individual element values unless at least t non-zero contributions are made at a given index. This mechanism inherently nullifies data reconstruction attacks, as it prevents the server from unmasking elements contributed by too few clients. The key idea is to introduce two types of new vectors: one for counting non-zero contributors and the other for masking model updates. In the SecAgg procedure, each client masks its model update using both conventional and newly introduced masks; at a given index, the new mask is sent to the server only if the number of contributors exceeds a certain

TABLE I
NOTATION FREQUENTLY USED IN THIS PAPER.

Notation	Description
$\mathcal{N}$	Set of all users
$\mathcal{C}, \mathcal{D}$	Set of clients and decryptors, respectively
δ_D	Dropout rate of decryptors
η_C, η_D	Colluding rate of clients and decryptors, respectively
$\boldsymbol{x}_i$	Model update vector of client $i \in \mathcal{C}$
$\mathcal{K}$	All indices in model update vector $\boldsymbol{x}_i$
t, ℓ	Threshold in SecAgg and secret sharing, respectively

threshold. Our approach does not rely on client-side model checks and remains simple, thereby avoiding the inherent limitations of existing methods.

The contributions of this paper are as follows. First, we propose *Per-element SecAgg*, a framework that reveals aggregated values only at indices with at least t non-zero contributions. Our design does not require any new cryptographic primitives beyond standard SecAgg protocols [4]–[6]. Second, we develop a full protocol by extending Flamingo [6], a lightweight SecAgg. Our protocol ensures that the guarantees of Per-element SecAgg hold even in the presence of a malicious server, user dropouts, and user collusion. Third, we implement the protocol and evaluate its performance. The results show that the additional overhead is acceptable, and that the impact of Per-element SecAgg on model performance is minimal, confirming that our protocol is practical.

The rest of the paper is organized as follows: Section II explains the preliminaries. Sections III and IV describe the design rationale of our Per-element SecAgg framework and the protocol design, respectively. Section V analyzes the cost and the security, and Section VI evaluates the performance of the protocol. Section VII summarizes the related work, and Section VIII concludes the paper.

II. PRELIMINARIES

In this section, we introduce the cryptographic primitives and the Flamingo SecAgg protocol, which are essential for understanding this paper. Table I summarizes the notation.

A. Cryptographic Primitives

1) Diffie-Hellman Key Exchange via PKI: A user i generates a private key $a_i \in \{0,1\}^\kappa$ and publishes the public key g^{a_i} via a public key infrastructure (PKI), where κ is a secyrity parameter and g is a generator of a cyclic group with prime order. A pair of users i, j establishes a shared secret $s_{i,j} = g^{a_i a_i}$ using their private key and the other's public key.

2) Pseudorandom Generators: $\mathsf{PRG}(r)$ is a deterministic function that expands a random seed $r \in \{0,1\}^\kappa$ into a longer pseudorandom string. The PRGs used in this paper are secure, meaning their outputs are computationally indistinguishable from truly random strings, as long as the seeds remain secret.

3) Pseudorandom Functions: $\mathsf{PRF}(k, x)$ is a deterministic function that maps an input x to a pseudorandom string of the same length, using a secret key $k \in \{0,1\}^\kappa$. The PRFs used in this paper are secure, meaning their outputs are computationally indistinguishable from truly random strings, as long as the keys remain secret.

4) Secret Sharing: Shamir's (ℓ, L) secret sharing scheme distributes a secret s using two algorithms: $\mathsf{SS.share}(s, \ell, L) \rightarrow \{\langle s \rangle_1, \ldots, \langle s \rangle_L\}$ splits s into L shares, and $\mathsf{SS.recon}(\{\langle s \rangle_l\}_{l \in \mathcal{L}}) \rightarrow s$ reconstructs s from any subset with $|\mathcal{L}| \geq \ell$. The scheme ensures that fewer than ℓ shares reveal no information about s.

B. Flaminngo SecAgg Protocol

SecAgg enables the server to obtain the sum of client inputs while keeping individual inputs private. We focus on the Flamingo [6] protocol because it fits seamlessly into our system model defined in Section III-A. A key innovation of Flamingo lies in its highly efficient unmasking phase, which is a bottleneck in [4]. Instead of involving all clients, Flamingo offloads this task to decryptors, a small set of users.

1) Setup Phase: The setup phase is executed only once before training begins. During this phase, each pair of users $i, j \in \mathcal{N}$ establishes a long-term secret $s_{i,j}$ and a symmetric key $k_{i,j}$ using Diffie-Hellman key exchange through PKI. Long-term secrets serve as the basis for generating pairwise masks in subsequent rounds. In addition, a trusted source of public randomness R [26] is used to select a small subset of users as decryptors $\mathcal{D}$ from $\mathcal{N}$.

2) Report Phase: In each round τ, each client $i \in \mathcal{C}$ uses R to determine a set of neighbors $\mathcal{A}_i \subset \mathcal{C}$ with whom it will compute pairwise masks. Using the long-term secrets, i derives round-specific pairwise seed $r_{i,j} \leftarrow \mathsf{PRF}(s_{i,j}, \tau)$ for all peers $j \in \mathcal{A}_i$. In addition, i randomly generates an individual seed r_i. Using these, i masks its model update vector $\boldsymbol{x}_i$ as follows:

$$[\![\boldsymbol{x}_i]\!] = \boldsymbol{x}_i + \underbrace{\mathsf{PRG}(r_i)}_{\text{(individual mask)}} + \underbrace{\sum_{j \in \mathcal{A}_i} \pm \mathsf{PRG}(r_{i,j})}_{\text{(pairwise mask)}}, \tag{1}$$

where the sign of each pairwise mask is positive if $i < j$ and negative otherwise. Although individual masks are essential for ensuring robustness against client dropouts, we assume no dropouts occur in this explanation for simplicity.

To enable unmasking of the individual mask, i secret-shares r_i as $\{\langle r_i \rangle_u\}_{u \in \mathcal{D}} \leftarrow \mathsf{SS.share}(r_i, \ell, |\mathcal{D}|)$. Then, it encrypts each share $\langle r_i \rangle_u$ using the symmetric key $k_{i,u}$ shared with decryptor u. That is, a total of $|\mathcal{D}|$ ciphertexts $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{u \in \mathcal{D}}$ are generated. Finally, i sends the masked input $[\![\boldsymbol{x}_i]\!]$ along with these ciphertexts to the server. Although there are $|\mathcal{D}|$ ciphertexts, each seed and its share are significantly smaller in size than the masked input vector itself, resulting in only marginal communication overhead.

3) Unmasking Phase: The server aggregates all the received masked vectors. Since we assume no client dropouts, the pairwise masks cancel out upon aggregation, yielding:

$$\sum_{i\in\mathcal{C}} [\![\boldsymbol{x}_i]\!] = \sum_{i\in\mathcal{C}} \boldsymbol{x}_i + \sum_{i\in\mathcal{C}} \mathsf{PRG}(r_i). \tag{2}$$

The goal of this phase is to remove individual masks so that the server can recover the plaintext $\sum_{i\in\mathcal{C}} \boldsymbol{x}_i$.

To this end, the server sends the ciphertexts of seed shares to their corresponding decryptors. Specifically, each decryptor $u \in \mathcal{D}$ receives a set of ciphertexts $\{[\![\langle r_i\rangle_u]\!]_{k_{i,u}}\}_{i\in\mathcal{C}}$, which it decrypts using the symmetric keys $\{k_{i,u}\}_{i\in\mathcal{C}}$. The plaintext seed shares are then returned to the server.

The server can reconstruct each seed r_i only if it collects at least ℓ shares from a subset of decryptors $\mathcal{D}_1 \subseteq \mathcal{D}$, i.e., $r_i \leftarrow \mathsf{SS.recon}(\{\langle r_i\rangle_u\}_{u\in\mathcal{D}_1})$, where $|\mathcal{D}_1| \geq \ell$. Finally, the server regenerates the individual masks by applying PRGs to each reconstructed seed and subtracts the total mask $\sum_{i\in\mathcal{C}} \mathsf{PRG}(r_i)$ from the aggregated vector to obtain the plaintext sum $\sum_{i\in\mathcal{C}} \boldsymbol{x}_i$.

III. Per-element SecAgg Framework

This section defines the system and threat models along with our design goals. Based on them, we propose the Per-element SecAgg framework against data reconstruction attacks.

A. System Model

We consider a star topology with a single server and a set of $\mathcal{N}$ users. Following prior work on Flamingo [6], Willow [8] and OPA [7], we assume two roles among users: clients and decryptors. In each round τ, a set of clients $\mathcal{C} \subset \mathcal{N}$ is selected at random based on a public source of randomness. Each client $i \in \mathcal{C}$ receives the global model from the server, trains it on its local dataset, and returns an update vector $\boldsymbol{x}_i \in \mathbb{R}^{|\mathcal{K}|}$. Separately, a set of decryptors $\mathcal{D} \subset \mathcal{N}$ is randomly selected using the same public randomness. Decryptors assist in the SecAgg procedure but do not participate in model training. The server receives only the aggregated sum of the update vectors $\sum_{i\in\mathcal{C}} \boldsymbol{x}_i$ via the SecAgg protocol, without learning any individual update.

Some users may drop out during the protocol due to unstable network conditions or battery limitations. We denote the dropout rate of decryptors in each round by δ_D. We do not explicitly define the dropout rate of clients, as their dropout does not affect our Per-element SecAgg mechanism.

B. Threat Model

We assume that the server behaves as a malicious adversary. It is allowed to arbitrarily deviate from the SecAgg protocol, including manipulation of protocol messages. In addition, the server can freely modify the global model distributed to clients in each round. The server may also collude with up to a fraction η_C of clients and a fraction η_D of decryptors, from whom it can obtain internal protocol information such as secret keys or secret shares. As with Flamingo, we assume that $\delta_D + \eta_D < 1/3$ and treat dropped decryptors and colluding decryptors as disjoint sets. The adversary's goal is to reconstruct the datasets of honest clients under SecAgg by launching sparse update-based attacks [9]–[14].

C. Design Goals

Our primary goal is to prevent data reconstruction attacks by simultaneously satisfying the following properties (P1)–(P4).

(P1) Per-element Threshold Aggregation. The server learns only the aggregated values of elements that are contributed by at least t honest clients. This property is the core of our work.

(P2) Security against a malicious server. Even if the server behaves maliciously, (P1) is still guaranteed. Specifically, this property considers the case where the malicious server sends forged information to honest decryptors.

(P3) Dropout Tolerance. Even if up to $\lfloor \delta_D \mathcal{D} \rfloor$ decryptors drop out, the protocol does not abort, and (P1) remains guaranteed. Tolerance to client dropouts is also required, but this can be addressed by existing SecAgg protocols [4]–[6] without affecting our framework. Therefore, the rest of this paper does not elaborate on recovery from client dropouts.

(P4) Collusion resilience. Even if the server colludes with up to $\lfloor \eta_C \mathcal{C} \rfloor$ clients and up to $\lfloor \eta_C \mathcal{D} \rfloor$ decryptors, (P1) is still guaranteed.

On the other hand, the following properties are not part of our goals.

Prevention of other attacks. We focus on defending against data reconstruction attacks, the most critical privacy threat in FL. Therefore, other attacks, such as property inference attacks [12] and label inference attacks [27], are out of scope.

Privacy of index information. Our approach reveals to the server and decryptors the indices where each client contributes non-zero values, as is commonly done for sparse FL [28]. While such exposure may raise privacy concerns, we do not attempt to hide them in our design, as it poses little threat. A detailed discussion is deferred to Section V-C.

D. Core Mechanism and Rationale

We introduce a lightweight mechanism that extends the SecAgg process to support Per-element SecAgg. This mechanism is built solely on standard cryptographic primitives (Section II-A). The key idea is to enable decryptors to control which elements are unmasked based on how many clients contribute non-zero values at each index.

The protocol proceeds as follows (illustrated in Fig. 1):

- Each client adds additional pairwise masks—shared with decryptors—to its local update vector, but only at the indices corresponding to non-zero elements.
- After collecting the masked vectors, the server counts how many clients contributed non-zero values at each index and forwards this information to the decryptors.
- For each index that exceeds threshold t, the decryptors return the corresponding pairwise mask, enabling the server to unmask the element of the aggregated vectors.

This overall flow is designed to satisfy our key security properties (P1) and (P2). In what follows, we explain the rationale behind each of the three components of the mechanism: 1)

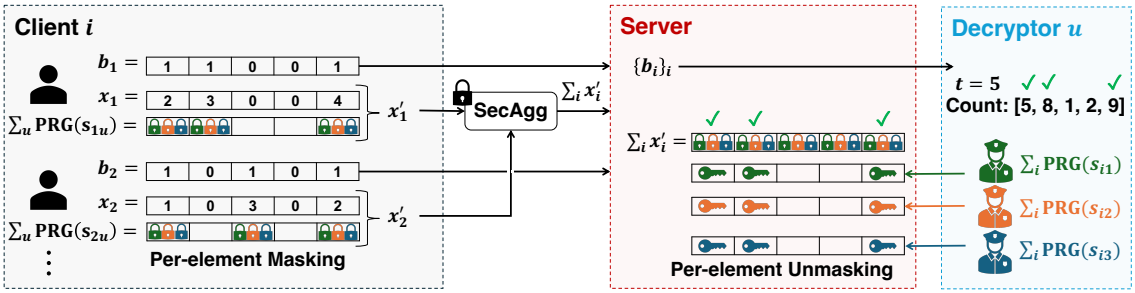

Fig. 1. Overview of Per-element SecAgg framework.

per-element contributor counting, 2) per-element masking, and 3) per-element unmasking. The overall protocol, along with the designs addressing (P3) and (P4), is presented in Sections IV and V, respectively.

1) Per-element Contributor Counting: To realize (P1), the decryptors must determine which indices have received at least t non-zero contributions. To this end, each client $i \in \mathcal{C}$ generates a binary indicator vector $\boldsymbol{b}_i$, where:

$$\boldsymbol{b}_i[k] = \begin{cases} 1 & \text{if } \boldsymbol{x}_i[k] \neq 0 \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

for all $k \in \mathcal{K}$. This vector flags the indices where the client's local update vector $\boldsymbol{x}_i$ has non-zero values. Clients send their indicator vectors to the server, which collects and forwards $\{\boldsymbol{b}_i\}_{i\in\mathcal{C}}$ to the decryptors, in accordance with the star topology of the network. This enables the decryptors to compute, for each index, the number of clients that have contributed non-zero values, and decide whether to unmask the corresponding element accordingly.

2) Per-element Masking: For (P1), the server forwards the set of indicator vectors $\{\boldsymbol{b}_i\}_{i\in\mathcal{C}}$ to the decryptors. However, a malicious server may forge these vectors, falsely claiming that an index k has received at least t non-zero contributions. This may cause honest decryptors to return masks for elements that do not genuinely satisfy the threshold condition, allowing the server to unmask them and thus violating (P2).

A straightforward solution would be to make each $\boldsymbol{b}_i$ verifiable using cryptographic tools. However, such an approach introduces additional cryptographic primitives and complicates the protocol. Instead, we adopt a simpler and more robust design: we tolerate the possibility of forged indicators, but ensure that any manipulation by the server cannot result in successful unmasking.

To this end, each client adds pairwise masks with the decryptors to its local update $\boldsymbol{x}_i$, but only at indices with non-zero values. As a result, the total mask at each index reflects only the contributions from clients with actual non-zero values at that index. Any mask derived from forged indicators fails to match this sum, making unmasking unsuccessful. Thus, only elements with contributions from at least t clients can be correctly unmasked. Server-client collusion resilience is addressed in Section V-B.

3) Per-element Unmasking: To enforce per-element unmasking control, each decryptor must return masks on an element-wise basis. A naive approach—used in Flamingo—would have decryptors send back the seeds (shares) allowing the server to regenerate the full mask efficiently. However, this would enable the server to unmask all elements, including those that should remain hidden, thus violating (P1).

To avoid this, we have each decryptor return a mask vector instead of the seeds. Specifically, the decryptor reveals actual mask values only for indices that meet the threshold condition. This design ensures that the server can unmask only the intended elements, preserving per-element confidentiality.

E. Efficiency Gains by Using Partial Vectors

We reduce the size of binary indicator vectors $\boldsymbol{b}_i$ and the computational and communication overhead of per-element masking/unmasking mechanisms.

1) Compression of Binary Indicator Vectors: In Section III-D, we design the protocol in which each client i sends its full binary indicator vector $\boldsymbol{b}_i$ to the server, which then forwards $\{\boldsymbol{b}_i\}_{i\in\mathcal{C}}$ to all decryptors. Since $\boldsymbol{b}_i$ has the same dimension as its local update $\boldsymbol{x}_i$ (i.e., $|\mathcal{K}|$), sending such large vectors is communication-inefficient. However, we can leverage the observation that local update vectors are typically sparse. According to [29], more than 95% of their elements are effectively zero. This means that $\boldsymbol{b}_i$ is also sparse. To reduce communication costs, we instead transmit the set $\mathcal{B}_i = \{k \in \mathcal{K} \mid \boldsymbol{b}_i[k] = 1\}$, which compresses the indicator information while preserving correctness.

2) Protecting Only FC Layers: We reduce the number of elements protected by masks by leveraging the observation that data reconstruction attacks primarily target a subset of fully connected (FC) layers, particularly those near the input or output layers [22], [23]. These gradients are semantically informative and less protected by architectural structures such as convolution or pooling. Therefore, we apply our mechanism only to the parts of $\boldsymbol{x}_i$ corresponding to selected FC layers. As detailed in Section VI-A2, this reduces the masking/unmasking scope to 5–40% of the entire update vector, depending on the model architecture, and significantly lowers both computational and communication overhead.

Setup Phase.
- **User** $i \in \mathcal{N}$
  - Generates key pairs $(g^{a_i}, a_i), (g^{b_i}, b_i)$ and stores its public keys (g^{a_i}, g^{b_i}) in PKI.
- **All:**
  - Given the security parameter κ, a public randomness R, the number of clients and decryptors, the size of neighbors, threshold values t, ℓ, and the parameter $\Delta_{\max}$.
  - Select the decryptors $\mathcal{D}$ with R.

Report Phase in round τ.
- **Client** $i \in \mathcal{C}$**:**
  - Select $\mathcal{A}_i$ with R.
  - Retrieve g^{a_j} for $j \in \mathcal{A}_i$, and g^{a_u} and g^{b_u} for $u \in \mathcal{D}$ from PKI.
  - Derive $s_{i,j} \leftarrow g^{a_i a_j}$ for $j \in \mathcal{A}_i$, and $s_{i,u} \leftarrow g^{a_i a_u}$, $k_{i,u} \leftarrow g^{b_i b_u}$ for $u \in \mathcal{D}$.
  - Compute $r_{i,j} \leftarrow \mathsf{PRF}(s_{i,j}, \tau)$ for $j \in \mathcal{A}_i$, and $r_{i,u} \leftarrow \mathsf{PRF}(s_{i,u}, \tau)$ for $u \in \mathcal{D}$.
  - Construct $\boldsymbol{b}_i$ and $\mathcal{B}_i$ from $\boldsymbol{x}_i$, and compute as: $\boldsymbol{x}'_i = \boldsymbol{x}_i + \boldsymbol{b}_i \odot \sum_{u \in \mathcal{D}} \mathsf{PRG}(r_{i,u})$.
  - Randomly generate r_i, and mask $\boldsymbol{x}'_i$ as: $[\![\boldsymbol{x}'_i]\!] = \boldsymbol{x}'_i + \mathsf{PRG}(r_i) + \sum_{j \in \mathcal{A}_i} \pm \mathsf{PRG}(r_{i,j})$.
  - Secret-share the seeds as $\{\langle r_i \rangle_u\}_{u \in \mathcal{D}} \leftarrow \mathsf{SS.share}(r_i, \ell, |\mathcal{D}|)$ and $\{\langle r_{i,v} \rangle_u\}_{u,v \in \mathcal{D}} \leftarrow \mathsf{SS.share}(\{r_{i,v}\}_{v \in \mathcal{D}}, \ell, |\mathcal{D}|)$.
  - Encrypt $\langle r_i \rangle_u$ and $\{\langle r_{i,v} \rangle_u\}_{v \in \mathcal{D}}$ with $k_{i,u}$ for each $u \in \mathcal{D}$ (i.e., $[\![\langle r_i \rangle_u]\!]_{k_{i,u}}$ and $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{v \in \mathcal{D}}$).
  - Send to the server: $[\![\boldsymbol{x}_i]\!]$, $\mathcal{B}_i$, $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{u \in \mathcal{D}}$, $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{u,v \in \mathcal{D}}$.
- **Server:**
  - Collect messages from clients $\mathcal{C}$ and aggregate $\sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!]$.
  - Send to decryptor $u \in \mathcal{D}$: $\{\mathcal{B}_i\}_{i \in \mathcal{C}}$ and $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{i \in \mathcal{C}}$.

Unmasking Phase in round τ.
- **Decryptor** $u \in \mathcal{D}$**:**
  - Retrieve g^{a_i} and g^{b_i} for $i \in \mathcal{C}$ from the PKI.
  - Derive $s_{i,u} \leftarrow g^{a_i a_u}$ and $k_{i,u} \leftarrow g^{b_i b_u}$ for $i \in \mathcal{C}$.
  - Compute $r_{i,u} \leftarrow \mathsf{PRF}(s_{i,u}, \tau)$ for $i \in \mathcal{C}$.
  - Construct $\mathcal{C}[k]$ for each k by using $\{\mathcal{B}_i\}_{i \in \mathcal{C}}$.
  - Derive $\boldsymbol{emk}_u[k]$ for each k: $\boldsymbol{emk}_u[k] = \sum_{i \in \mathcal{C}[k]} \mathsf{PRG}(r_{i,u})[k]$ if $|\mathcal{C}[k]| \geq t$; otherwise, $\perp$.
  - Decrypt $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{i \in \mathcal{C}}$ with $\{k_{i,u}\}_{i \in \mathcal{C}}$.
  - Send to the server: $\boldsymbol{emk}_u$ and $\{\langle r_i \rangle_u\}_{i \in \mathcal{C}}$.
- **Server:**
  - Collect messages from decryptors $\mathcal{D}_1 \subseteq \mathcal{D}_2$.
  - If $|\mathcal{D}_1| \geq \ell$, reconstruct seeds: $\{r_i\}_{i \in \mathcal{C}} \leftarrow \mathsf{SS.recon}(\{\{\langle r_i \rangle_u\}_{u \in \mathcal{D}_1}\}_{i \in \mathcal{C}})$; otherwise, abort.
  - Subtract $\sum_{u \in \mathcal{D}_1} \boldsymbol{emk}_u$ and $\sum_{i \in \mathcal{C}} \mathsf{PRG}(r_i)$ from $\sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!]$.
  - If $\mathcal{D}_1 \neq \mathcal{D}$, derive $\mathcal{V} = \mathcal{D} \setminus \mathcal{D}_1$.
  - Send to the decryptor $u \in \mathcal{D}_1$: $\mathcal{V}$ and $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{i \in \mathcal{C}, v \in \mathcal{V}}$

Dropout Recovery Phase in round τ (if $\mathcal{D}_1 \neq \mathcal{D}$).
- **Decryptor** $u \in \mathcal{D}_1$**:**
  - Proceed only if $|\mathcal{V}| \geq \Delta_{\max}$; otherwise, abort.
  - Decrypt $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{v \in \mathcal{V}}$. with $\{k_{i,u}\}_{i \in \mathcal{C}}$.
  - Send to the server: $\{\langle r_{i,v} \rangle_u\}_{i \in \mathcal{C}, v \in \mathcal{V}}$.
- **Server:**
  - Collect messages from decryptors $\mathcal{D}_2 \subseteq \mathcal{D}_1$.
  - If $|\mathcal{D}_2| \geq \ell$, reconstruct seeds: $\{r_{i,u}\}_{i \in \mathcal{C}, u \in \mathcal{D} \setminus \mathcal{D}_1} \leftarrow \mathsf{SS.recon}(\{\langle r_{i,u} \rangle_v\}_{i \in \mathcal{C}, v \in \mathcal{D} \setminus \mathcal{D}_1})$; otherwise, abort.
  - Construct $\{\boldsymbol{b}_i\}_{i \in \mathcal{C}}$ from $\{\mathcal{B}_i\}_{i \in \mathcal{C}}$.
  - Subtract $\sum_{i \in \mathcal{C}} \sum_{u \in \mathcal{D} \setminus \mathcal{D}_1} \boldsymbol{b}_i \odot \mathsf{PRG}(r_{i,u})$ from $\sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!] - \sum_{u \in \mathcal{D}_1} \boldsymbol{emk}_u$.

Fig. 2. The full protocol that realizes Per-element SecAgg based on Flamingo. Red-colored components indicate the extended procedures. For simplicity, client dropouts are not considered.

IV. Protocol

We present the full protocol that realizes Per-element SecAgg. It extends Flamingo by incorporating the mechanisms explained in Section III for (P1) and (P2), and introduces an additional phase for dropout tolerance of decryptors (P3).

A. Protocol Covering for (P1) and (P2)

This subsection explains the protocol phases—mainly the Report Phase and Unmasking Phase—that work together to achieve (P1) and (P2). We focus on the red-colored parts in Fig. 2, which explain the extensions for Per-element SecAgg. For simplicity, client dropout recovery mechanisms are omitted here, but they can be achieved using Flamingo's original techniques [6] without affecting our extension.

1) Setup Phase: This phase is identical to the Setup Phase in Flamingo (Section II-B). Our mechanism requires no additional setup beyond what Flamingo already provides.

2) Report Phase: To enable additional masking for Per-element SecAgg, each client $i \in \mathcal{C}$ first retrieves the public keys of all decryptors $u \in \mathcal{D}$ from the PKI and derives long-term secrets $s_{i,u} \leftarrow g^{a_i a_u}$. Then, i computes a round-specific seed $r_{i,u} \leftarrow \mathsf{PRF}(s_{i,u}, \tau)$.

i constructs an indicator vector $\boldsymbol{b}_i$ and the set of its non-zero indices $\mathcal{B}_i$ from its local update $\boldsymbol{x}_i$, following Eq. (3). Next, for per-element masking described in Section III-D2, i masks only the non-zero elements of $\boldsymbol{x}_i$ as follows:

$$\boldsymbol{x}'_i = \boldsymbol{x}_i + \boldsymbol{b}_i \odot \sum_{u \in \mathcal{D}} \mathsf{PRG}(r_{i,u}), \tag{4}$$

where $\odot$ denotes element-wise multiplication. The masked vector $\boldsymbol{x}'_i$ is then further processed using Flamingo's original masking procedure—adding an individual mask and pairwise masks—to produce the final masked vector $[\![\boldsymbol{x}'_i]\!]$.

To prepare for decryptor dropout recovery (explained in the next subsection), i secret-shares each seed $r_{i,u}$ across all decryptors: $\{\langle r_{i,v} \rangle_u\}_{u,v \in \mathcal{D}} \leftarrow \mathsf{SS.share}(\{r_{i,v}\}_{v \in \mathcal{D}}, \ell, |\mathcal{D}|)$. Then, it encrypts each share $\langle r_{i,v} \rangle_u$ using the symmetric key $k_{i,u}$, producing $[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}$. Finally, i sends the following to the server: $[\![\boldsymbol{x}'_i]\!]$, $\mathcal{B}_i$, and the encrypted shares $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{u \in \mathcal{D}}$ and $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{u,v \in \mathcal{D}}$.

Upon collecting all messages from the clients, the server aggregates the masked vectors:

$$\sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!] = \sum_{i \in \mathcal{C}} \boldsymbol{x}_i + \sum_{i \in \mathcal{C}} \mathsf{PRG}(r_i) + \sum_{i \in \mathcal{C}} \sum_{u \in \mathcal{D}} \mathsf{PRG}(r_{i,u}). \tag{5}$$

To unmask Eq. (5), the server forwards $\{[\![\langle r_i \rangle_u]\!]_{k_{i,u}}\}_{u \in \mathcal{D}}$ and $\{\mathcal{B}_i\}_{i \in \mathcal{C}}$ to each decryptor $u \in \mathcal{D}$. This completes the forwarding step described in Section III-D1, allowing the decryptors to compute per-element contributor counts.

3) Unmasking Phase: Each decryptor $u \in \mathcal{D}$ first derives round-specific seeds $\{r_{i,u}\}_{i \in \mathcal{C}}$ using the public keys of clients and their long-term secrets. Next, using the collection $\{\mathcal{B}_i\}_{i \in \mathcal{C}}$ received from the server, u constructs the set of contributors for each index $k \in \mathcal{K}$ as $\mathcal{C}[k] = \{i \in \mathcal{C} \mid k \in \mathcal{B}_i\}$. This identifies the clients who reported non-zero values at index k. Based on the contributor count $|\mathcal{C}[k]|$, u determines which indices satisfy the threshold t and generates an element-wise mask vector $\boldsymbol{emk}_u$ as follows:

$$\boldsymbol{emk}_u[k] = \begin{cases} \sum_{i \in \mathcal{C}[k]} \mathsf{PRG}(r_{i,u})[k] & \text{if } |\mathcal{C}[k]| \geq t \\ \perp & \text{otherwise.} \end{cases} \tag{6}$$

where $\perp$ denotes a null value and $\mathsf{PRG}(r_{i,u}[k])$ denotes the k-th element of the pseudorandom mask vector. In this way, u explicitly controls which indices are eligible for unmasking, in accordance with the per-element threshold policy described in Section III-D3. Finally, the decryptor returns $\boldsymbol{emk}_u$ and the decrypted seed shares $\{\langle r_i \rangle_u\}_{i \in \mathcal{C}}$ to the server.

Upon receiving messages from all decryptors, the server reconstructs the individual seeds and regenerates $\sum_{i \in \mathcal{C}} \mathsf{PRG}(r_i)$. Then, using the sum of element-wise masks $\sum_{u \in \mathcal{D}} \boldsymbol{emk}_u$, the server performs the unmasking:

$$\begin{aligned} \boldsymbol{y} &= \sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!] - \sum_{i \in \mathcal{C}} \mathsf{PRG}(r_i) - \sum_{u \in \mathcal{D}} \boldsymbol{emk}_u \\ &= \begin{cases} \sum_{i \in \mathcal{C}} \boldsymbol{x}_i[k] & \text{if } |\mathcal{C}[k]| \geq t \\ \perp & \text{otherwise.} \end{cases} \end{aligned} \tag{7}$$

This ensures that only elements with at least t contributors are unmasked, thereby fulfilling (P1).

B. Protocol Covering for (P3)

This subsection extends the protocol presented in Section IV-A to tolerate decryptor dropouts (P3). We begin by explaining the motivation and rationale behind this extension. We then identify a potential threat to (P2) that arises from this extension and describe our solution. Finally, we present the detailed protocol steps.

1) Motivation: In the Unmasking Phase explained in Section IV-A3, let $\mathcal{D}_1 \subseteq \mathcal{D}$ be the set of decryptors from whom the server receives element-wise masks $\boldsymbol{emk}_u$. If any decryptor drops out (i.e., $\mathcal{D}_1 \neq \mathcal{D}$), the corresponding element-wise masks $\{\boldsymbol{emk}_u\}_{u \in \mathcal{D} \setminus \mathcal{D}_1}$ are missing. As a result, the unmasking in Eq. (7) fails for all elements due to incomplete subtraction. To satisfy (P3), we introduce a dropout recovery mechanism that is executed only when $\mathcal{D}_1 \neq \mathcal{D}$.

2) Rationale: To tolerate decryptor dropouts, clients provide encrypted shares of all seeds in advance during the Report Phase. If some decryptors drop out, the server sends the drop list $\mathcal{V} = \mathcal{D} \setminus \mathcal{D}_1$ and corresponding ciphertexts of seed shares to the surviving decryptors $\mathcal{D}_1$, requests the plaintexts, and reconstructs the missing seeds. Since only seed shares are exchanged, the communication overhead remains marginal.

This strategy may appear to contradict the per-element unmasking (Section III-D3), which avoids disclosing seeds to the server. However, revealing seeds for the dropped decryptors does not violate (P1), since unmasking depends on element-wise masks $\boldsymbol{emk}_u$ from honest decryptors in $\mathcal{D}_1$, which are revealed only for elements meeting the threshold. Thus, the server still cannot unmask under-contributed elements.

3) Risk of Violating (P2) and Its Solution: The dropout recovery mechanism poses a threat to (P2), in which a malicious server may add surviving and honest decryptors to the dropout list $\mathcal{V}$, so that they are disguised as having dropped out. The server can then obtain at least ℓ shares for all decryptors' seeds and fully unmask every element, violating (P1).

To counter this threat, we adopt a simple but practical strategy that requires no additional round for checking dropout decryptors. We introduce a global constant $\Delta_{\max}$, which defines the maximum number of decryptor dropouts tolerated by honest decryptors. If $|\mathcal{V}| > \Delta_{\max}$, the protocol is aborted by the honest decryptors. This bound limits how many seed shares the server can receive from the honest decryptors, ensuring that at least one honest decryptor's seed cannot be recovered. As proven in Section V-B, setting $\Delta_{\max} = \lceil \ell/2 \rceil$ (where ℓ is the threshold of the secret sharing scheme) prevents this risk while ensuring that the protocol does not abort due to decryptor dropouts.

4) Procedure: Upon detecting a decryptor dropout (i.e., $\mathcal{D}_1 \neq \mathcal{D}$) during the Unmasking Phase, the server constructs a list $\mathcal{V} = \mathcal{D} \setminus \mathcal{D}_1$ of the dropped decryptors. It then sends $\mathcal{V}$ and the encrypted seed shares $\{[\![\langle r_{i,v} \rangle_u]\!]_{k_{i,u}}\}_{i \in \mathcal{C}, v \in \mathcal{V}}$ to each surviving decryptor $u \in \mathcal{D}_1$.

In the Dropout Recovery Phase, each decryptor $u \in \mathcal{D}_1$ first verifies that $|\mathcal{V}| \leq \Delta_{\max}$. Then, u decrypts the received ciphertexts using the symmetric keys $\{k_{i,u}\}_{i \in \mathcal{C}}$, and returns the plaintext shares to the server.

The server collects responses from a set $\mathcal{D}_2$ of at least ℓ decryptors and reconstructs all missing seeds $\{r_{i,v}\}_{i \in \mathcal{C}, v \in \mathcal{V}}$ via secret sharing. Using the reconstructed seeds and the indicator vectors $\{\boldsymbol{b}_i\}_{i \in \mathcal{C}}$, the server completes the unmasking step:

$$\begin{aligned} \boldsymbol{y} = \sum_{i \in \mathcal{C}} [\![\boldsymbol{x}'_i]\!] \; - \; \sum_{i \in \mathcal{C}} \mathsf{PRG}(r_i) \\ - \sum_{u \in \mathcal{D}_1} \boldsymbol{emk}_u \; - \sum_{v \in \mathcal{D} \setminus \mathcal{D}_1} \boldsymbol{b}_i \odot \mathsf{PRG}(r_{i,v}). \end{aligned} \tag{8}$$

This yields the correct aggregate $\boldsymbol{y}$, and achieves (P3).

TABLE II
COST PER PROTOCOL PHASE OF OUR PROTOCOL

	Client	Server
	Computation	
Report	$\mathcal{O}(DK' + AK + D^3)$	$\mathcal{O}(CK)$
Unmask	$\mathcal{O}(CK')$	$\mathcal{O}(DK' + CD^2 + CK)$
DropRcv	$\mathcal{O}(CV)$	$\mathcal{O}(V(CD^2 + K'))$
	Communication	
Report	$\mathcal{O}(K + \alpha K' + D^2)$	$\mathcal{O}(C(D^2 + DK'\alpha + K))$
Unmask	$\mathcal{O}(CK'\alpha)$	$\mathcal{O}(D(K' + CV))$
DropRcv	$\mathcal{O}(CV)$	$\mathcal{O}(CDV)$

Notation: C: number of clients, D: number of decryptors, A: size of neighbors of each client, V: number of dropped decryptors, K: vector size of $\boldsymbol{x}_i$, K': vector size to be additionally masked, α: sparsity of a vector.

V. COST, SECURITY AND PRIVACY ANALYSIS

A. Computation and Communication Cost Analysis

The computation and communication costs of our protocol are summarized in Table II. We define the computation cost by treating the following operations as $\mathcal{O}(1)$: one PRF evaluation, one PRG output for a vector element, one symmetric

encryption/decryption, and one addition/subtraction of a vector element. The cost of secret sharing (both SS.share and SS.recon) is modeled as $\mathcal{O}(|D|^2)$ per secret.

For communication cost, we define $\mathcal{O}(1)$ as the transfer of one vector element, one item in a set, or one share.

B. Security Analysis

We analyze the security of our protocol. We first show how collusion resilience (P4) is achieved by appropriate parameter settings. Finally, we prove that our protocol remains secure even in the presence of a malicious server, colluding users, and decryptor dropouts, thereby satisfying (P1)–(P4).

1) Parameter Settings For (P4): As shown in Eq. (6), each decryptor returns its element-wise mask if the number of non-zero contributors at an index reaches the threshold t. However, the element-wise mask alone cannot ensure (P1)—that the server learns only the aggregated values of elements contributed by at least t honest clients—in the presence of colluding clients. This is because colluding clients may maliciously flag their binary indicator vectors, making the contribution count exceed t.

To address this vulnerability, we redefine the decryptor's threshold from t to t' and discuss how to set it to enforce (P1) under collusion. Clearly, $t' > \lfloor \eta_C |\mathcal{C}| \rfloor$ is required to prevent colluding clients from meeting the threshold alone. The goal of setting t' is to hide the element values unless at least t honest clients contribute. We now define how to set t' to ensure this.

Theorem 1. *Let t be the minimum number of honest client contributions required for an element to be revealed. Let t' be the threshold used by the decryptors to determine whether to return the element-wise mask for unmasking. Suppose the server may collude with up to $\lfloor \eta_C |\mathcal{C}| \rfloor$ clients. Then, setting $t' = \lfloor \eta_C |\mathcal{C}| \rfloor + t$ ensures that no element is unmasked unless at least t honest clients have contributed non-zero values.*

Sketch Proof. Since the server can control up to $\lfloor \eta_C |\mathcal{C}| \rfloor$ clients, it may attempt to inflate the contribution count to $\lfloor \eta_C |\mathcal{C}| \rfloor$ at arbitrary indices. By setting $t' = \lfloor \eta_C |\mathcal{C}| \rfloor + t$, the server with colluding clients cannot inflate the count to t'. □

We then discuss the selection of ℓ and $\Delta_{\max}$ to achieve resilience against colluding decryptors.

Theorem 2. *Let $\mathcal{D}_1 \subseteq \mathcal{D}$ be the set of online decryptors in the Unmasking Phase, and $\mathcal{D}_2 \subseteq \mathcal{D}_1$ be those who respond in the Dropout Recovery Phase. Suppose the server may collude with up to $\lfloor \eta_D |\mathcal{D}| \rfloor$ decryptors and up to $\lfloor \delta_D |\mathcal{D}| \rfloor$ honest decryptors may drop out, under the constraint $\delta_D + \eta_D < 1/3$. Then, setting $\ell = \lfloor 2|\mathcal{D}|/3 \rfloor + 1$, $\Delta_{\max} = \lceil \ell/2 \rceil$ simultaneously ensures the following:*
(Recovery) The server can reconstruct the seeds of all dropped decryptors $v \in \mathcal{D} \setminus \mathcal{D}_1$ if $|\mathcal{D}_2| \geq \ell$.
(Security) A malicious server cannot reconstruct the complete seeds of all ($\lfloor (1-\eta_D)|\mathcal{D}| \rfloor$) honest decryptors.

Sketch Proof. Each decryptor $u \in \mathcal{D}_2$ returns $|\mathcal{C}|$ seed shares for each dropped decryptor $v \in \mathcal{V}$ reported by the server. Treating this response as one unit per v, the total number of shares the server receives is $\Delta_{\max} \cdot \lfloor (1 - \delta_D - \eta_D)|\mathcal{D}| \rfloor$.

(Recovery) To reconstruct the seeds of $\lfloor \delta_D |\mathcal{D}| \rfloor$ dropped decryptors, the server requires $\lfloor \delta_D |\mathcal{D}| \rfloor \cdot (\ell - \lfloor \eta_D |\mathcal{D}| \rfloor)$ shares. The server can successfully recover the missing seeds if the number of received shares is at least this amount, i.e., $\Delta_{\max} \cdot \lfloor (1 - \delta_D - \eta_D)|\mathcal{D}| \rfloor \geq \lfloor \delta_D |\mathcal{D}| \rfloor \cdot (\ell - \lfloor \eta_D |\mathcal{D}| \rfloor)$. In the worst-case setting for recovery, where $\delta_D \to 1/3$ and $\eta_D \to 0$, this simplifies to $\Delta_{\max} \geq \ell/2$.

(Security) To reconstruct all seeds of $\lfloor (1 - \eta_D)|\mathcal{D}| \rfloor$ honest decryptors, a malicious server requires $\lfloor (1 - \eta_D)|\mathcal{D}| \rfloor \cdot (\ell - \lfloor \eta_D |\mathcal{D}| \rfloor)$ shares. To ensure security, the number of shares the server obtains must be strictly less than this quantity: $\Delta_{\max} \cdot \lfloor (1 - \delta_D - \eta_D)|\mathcal{D}| \rfloor < \lfloor (1 - \eta_D)|\mathcal{D}| \rfloor \cdot (\ell - \lfloor \eta_D |\mathcal{D}| \rfloor)$. In the worst-case for security, where $\delta_D \to 0$ and $\eta_D \to 1/3$, this simplifies to $\Delta_{\max} < \ell - |\mathcal{D}|/3$.

From both conditions, $\Delta_{\max}$ is $\ell/2 \leq \Delta_{\max} < \ell - |\mathcal{D}|/3$ is derived. This range is non-empty if $\ell > 2|\mathcal{D}|/3$. Accordingly, the optimal parameters that satisfy both recovery and security are given by $\ell = \lfloor 2|\mathcal{D}|/3 \rfloor + 1$ and $\Delta_{\max} = \lceil \ell/2 \rceil$. □

2) Per-element Threshold Aggregation: We now show that our protocol ensures per-element threshold aggregation.

Theorem 3. *Let the parameters be $t' = \lfloor \eta_C |\mathcal{C}| \rfloor + t$, $\ell = \lceil 2|\mathcal{D}|/3 \rceil + 1$, and $\Delta_{\max} = \lfloor \ell/2 \rfloor$. Suppose the server colludes with up to $\lfloor \eta_C |\mathcal{C}| \rfloor$ clients and $\lfloor \eta_D |\mathcal{D}| \rfloor$ decryptors, and up to $\lfloor \delta_D |\mathcal{D}| \rfloor$ decryptors may drop out, with $\delta_D + \eta_D < 1/3$. Assume the underlying cryptographic primitives, PRG-based masking encryption [30] and the Flamingo protocol are secure. Then, the server learns only the sum of elements contributed by at least t honest clients; elements with fewer contributions remain hidden.*

Sketch Proof. A full proof via a hybrid argument is omitted; we sketch the key ideas. By the security of Flamingo, the server receives only the masked sum $\sum_{i \in \mathcal{C}} \boldsymbol{x}'_i$. This holds even if the server is semi-honest or malicious.

If the number of contributors at an index is at least t, then decryptors return the corresponding element-wise masks. These are sufficient to cancel out all per-element masking, allowing the server to recover the true sum at that index. For indices with fewer than t contributors, at least one honest decryptor will withhold its mask. Due to the unpredictability of PRG outputs and the security of the underlying key agreement, this missing mask renders the masked value computationally indistinguishable from random.

Theorems 1 and 2 guarantee that the server cannot forge contribution counts or manipulate dropout reports to gain access to missing masks. Thus, elements are revealed if and only if at least t clients contribute non-zero values. □

C. Privacy Analysis

Our protocol reveals to the server and decryptors the indices at which each client has non-zero updates, via the set of indices $\mathcal{B}_i$. This subsection discusses the potential privacy implications of this design choice.

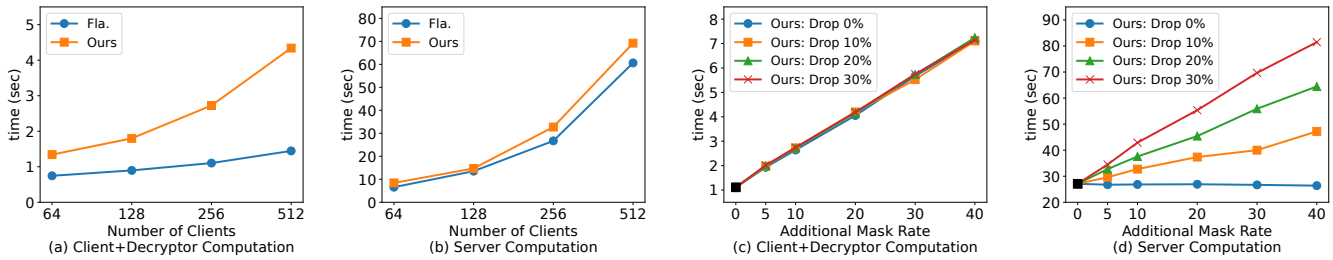

Fig. 3. Computation time vs. number of clients and masking ratio (model update vector dimension is 5M, number of decryptors is 40).

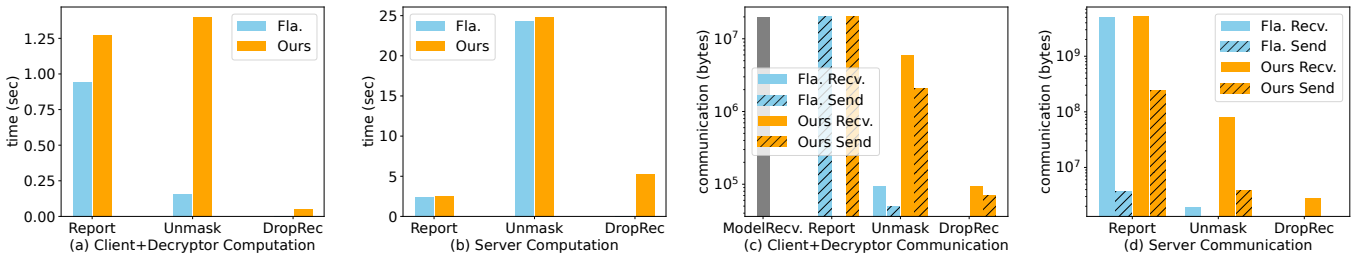

Fig. 4. Breakdown of protocol overhead in each phase (model update vector dimension is 5M, number of clients is 256, number of decryptors is 40, and dropout rate of decryptors is 10%).

One concern is that index exposure may lead to data reconstruction. Us Sami et al. [31] argue that, under SecAgg, knowing the indices of non-zero updates can allow the server to reconstruct clients' datasets by solving linear systems. However, this attack assumes the server sends an unchanged global model to the same client for hundreds of rounds, which is unrealistic in our setting, where clients are randomly reselected in each round (see Section III-A).

Another possible risk is attribute inference. Pasquini et al. [12] demonstrate that, by observing which indices are non-zero, the server can infer sensitive properties of a client's dataset. As noted in Section III-C, our framework specifically targets data reconstruction attacks and does not protect against such inference. Addressing this remains future work.

VI. Evaluation

We compare our proposed protocol with Flamingo and evaluate the additional overhead. We also examine the impact of Per-element SecAgg on model performance.

A. Experimental Setup

1) Implementation: We implement our protocol on the ABIDES simulator [32], which has been used to evaluate several SecAgg protocols [6], [7], [21]. Cryptographic primitives are implemented by using the same modules used in Flamingo.

2) Parameter Settings: To reflect practical FL scenarios in which a malicious server may launch data reconstruction attacks under SecAgg, we configure the parameters as follows.

Model Update Vector Dimensionality. We assume a cross-device setting in which clients are resource-constrained user devices such as smartphones. In such settings, models typically contain several million parameters (e.g., ResNet-9 has 4.9M, LSTM has 8.3M [33], and small Transformer models around 4.1M [34]). Based on this observation, we fix the dimensionality of the model update vector to 5M for all evaluations.

Additional Mask Rate As shown in Section V-A, the overhead of our Per-element SecAgg depends on the proportion of the update vector that is masked. When data reconstruction attacks rely solely on model parameter manipulation, the masked portion typically remains below 10% for architectures such as ResNet and Transformer (e.g., Wen et al. [10] propose an attack that observes only the output layer, which accounts for 4.4% of the parameters in ResNet-18 [35]). However, attacks that involve modifications to the model architecture—such as inserting additional linear layers for reconstruction [13]—may require a higher masking rate. This also holds for potential future attacks that exploit deeper model structures. Therefore, we fix the update vector size to 5M and vary the masking ratio from 5% to 40% in our evaluation.

Sparsity of update vectors. To prevent reconstruction attacks, elements in the model update vector with magnitudes below a threshold λ should be treated as zeros. This is aligned with threshold-based sparsification approaches in FL [29], [36]. According to [29], setting λ between 10^{-3} and 10^{-2} results in over 99% of elements being zeroed out. In our evaluation, we conservatively assume a sparsity level of 95%.

B. Computation Time and Communication Overhead

1) When Additional Mask Rate is 10%: We evaluate the computation time and communication overhead in the most practical setting, where the additional mask rate is 10%. Figs. 3a and 3b show the computation time for a user (client + decryptor) and the server, respectively, as the number of clients ($|\mathcal{C}|$) increases, in comparison with Flamingo ("Fla.") and our protocol with Per-element SecAgg ("Ours").

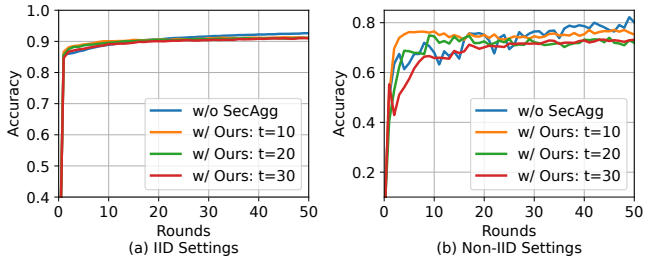

Fig. 5. Accuracy under MNIST dataset (number of clients is 100, the optimizer is FedAvg, number of local training epochs in each client is 5).

At 256 clients, the additional time compared to Flamingo is 1.6s on the user side and 6.0s on the server side, totaling just 7.6s per round. While both increase with $|\mathcal{C}|$, the overhead remains modest. Fig. 4 breaks down the computation time and communication overhead at 256 clients. On the user side, 1.2s (75%) of the overhead comes from the decryptor's Unmasking phase (Fig. 4a), which is acceptable, as decryptors do not perform training and remain lightweight. On the server side, 5.3s (80%) stems from Dropout Recovery (Fig. 4b).

Figs. 4c and 4d show the communication overheads, with increases of 1.21× on the user side and 1.07× on the server side. User-side growth is more notable. This is mainly due to the decryptor's reception of the index sets $\{\mathcal{B}_i\}_{i\in\mathcal{C}}$ ("Recv." in Unmask) and the transmission of element-wise masks ("Send" in Unmask), which together account for 96% of the increase. Nevertheless, this is acceptable, as both are much lighter than client communication, such as downloading the global model ("Model Recv.") or uploading updates ("Send" in Report).

2) Computation Time under Varying Additional Mask Rate: Figs. 3c and 3d show the computation time for a user and the server, respectively, as the additional mask rate varies. Note that the case of 0% corresponds to the original Flamingo. We also vary the decryptor dropout rate across 0%–30%.

On the user side, computation time increases roughly linearly with the additional mask rate. The maximum overhead is a 6.4× increase over Flamingo when 40% of elements are additionally masked. The user time is largely unaffected by decryptor dropout, since the corresponding recovery involves only lightweight symmetric decryption.

The server-side computation time also grows linearly with the additional mask rate, and the slope increases with higher dropout rates. The highest overhead—2.9× over Flamingo—occurs at 40% additional masking and 30% dropout. Notably, when the dropout rate is 0%, the server time remains nearly identical to Flamingo, regardless of the additional mask rate. This is because, in the absence of dropout, the server only performs element-wise subtraction on masks, a lightweight operation as shown in Fig. 4b.

C. Impact on Model Performance

In Per-element SecAgg, elements with fewer than t non-zero contributions in a round are not revealed to the server, and the corresponding elements in the global model cannot be updated. While this design enhances privacy—especially with larger t, it may affect model accuracy.

We evaluate this effect using a three-layer fully connected neural network and the MNIST dataset [21], with 100 clients. Model updates are sparsified by thresholding small-magnitude elements, yielding about 95% sparsity. In the IID setting, each client holds data uniformly sampled across all 10 labels. In the non-IID setting, each client holds data from 2 labels.

Under the IID setting (Fig. 5a), Per-element SecAgg shows almost the same convergence speed and final accuracy as the baseline ("w/o SecAgg"), even when t is increased; the final accuracy difference remains as small as 0.01. Under the non-IID setting (Fig. 5b), the accuracy transition becomes less stable due to data heterogeneity, and when $t = 20, 30$, the final accuracy degrades by up to 0.05 compared to the baseline.

This is likely due to reduced overlap of non-zero indices. As noted in [37], [38], IID clients are more likely to update the same elements, making threshold satisfaction easier—even when the updates are highly sparse. In contrast, under non-IID settings, heterogeneous data distributions lead to diverging update patterns among clients. This divergence, combined with high update sparsity, makes the overlap of non-zero indices especially low, which in turn significantly hinders the server's ability to collect at least t contributions at each index. Improving robustness in non-IID settings under high t remains an important direction for future work.

VII. Related Work

This section summarizes data reconstruction attacks that exploit the sparsity of model update vectors, which our Per-element SecAgg can potentially prevent. Existing countermeasures and their limitations are already discussed in Section I.

A common strategy exploits the ReLU activation, which outputs zero for negative inputs. A malicious server can manipulate model weights so that only a specific sample yields a non-zero output. When distributed to a client, the resulting update reveals gradient information corresponding to that sample, as demonstrated by Pasquini et al. [12] and Boenisch et al. [11]. Other approaches aim for more accurate reconstruction by modifying not only model parameters but also the architecture itself. Fowl et al. [9] and Zhao et al. [13] propose attacks that insert linear layers with ReLU activation into the global model to directly extract data from the corresponding updates. Wen et al. [10] amplify gradients only for target classes or features by carefully altering weights and biases. Since this attack isolates updates by feature rather than by client, distributing the same manipulated model to all clients can still enable data extraction, effectively bypassing SecAgg. Although most existing attacks target image reconstruction, Chu et al. [14] extend this to text. They focus on Transformer models and amplify updates associated with specific keywords by modifying multi-head self-attention weights, enabling the reconstruction of inputs such as credit card numbers.

VIII. Conclusion

This paper proposes Per-element SecAgg, a new mechanism that enhances SecAgg in FL by revealing aggregated values

only when at least t clients contribute non-zero elements. We design a concrete protocol by integrating this mechanism into the Flamingo SecAgg protocol without introducing new cryptographic primitives. We evaluate the additional overhead and model performance impact of Per-element SecAgg, demonstrating its practicality in realistic settings.

REFERENCES

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. Artificial Intelligence and Statistics (AISTATS)*, 2017.

[2] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi, "Beyond inferring class representatives: User-level privacy leakage from federated learning," in *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, 2019.

[3] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients - how easy is it to break privacy in federated learning?" in *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[4] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2017.

[5] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2020.

[6] Y. Ma, J. Woods, S. Angel, A. Polychroniadou, and T. Rabin, "Flamingo: Multi-round single-server secure aggregation with applications to private federated learning," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2023.

[7] H. Karthikeyan and A. Polychroniadou, "OPA: One-shot private aggregation with single client interaction and its applications to federated learning," in *Proc. Crypto*, 2025.

[8] J. Bell-Clark, A. Gascón, B. Li, M. Raykova, and P. Schoppmann, "Willow: Secure aggregation with one-shot clients," in *Proc. Crypto*, 2025.

[9] L. H. Fowl, J. Geiping, W. Czaja, M. Goldblum, and T. Goldstein, "Robbing the fed: Directly obtaining private data in federated learning with modified models," in *Proc. International Conference on Learning Representations (ICLR)*, 2022.

[10] Y. Wen, J. A. Geiping, L. Fowl, M. Goldblum, and T. Goldstein, "Fishing for user data in large-batch federated learning via gradient magnification," in *Proc. International Conference on Machine Learning (ICML)*, vol. 162, 2022.

[11] F. Boenisch, A. Dziedzic, R. Schuster, A. S. Shamsabadi, I. Shumailov, and N. Papernot, "When the curious abandon honesty: Federated learning is not private," in *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.

[12] D. Pasquini, D. Francati, and G. Ateniese, "Eluding secure aggregation in federated learning via model inconsistency," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2022.

[13] J. C. Zhao, A. Sharma, A. R. Elkordy, Y. H. Ezzeldin, S. Avestimehr, and S. Bagchi, "Loki: Large-scale data reconstruction attack against federated learning through model manipulation," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2024.

[14] H.-M. Chu, J. Geiping, L. H. Fowl, M. Goldblum, and T. Goldstein, "Panning for gold in federated learning: Targeted text extraction under arbitrarily large-scale aggregation," in *Proc. International Conference on Learning Representations (ICLR)*, 2023.

[15] N. H. Sultan, Y. Bo, Y. Gao, S. Camtepe, A. Mahboubi, H. T. Bui, A. Chauhan, H. Aboutorab, M. Bewong, P. Gauravaram, R. Islam, and S. Abuadbba, "RLSA-PFL: Robust lightweight secure aggregation with model inconsistency detection in privacy-preserving federated learning," *arXiv preprint arXiv:2502.08989*, 2025.

[16] Z. Tan, J. Le, F. Yang, M. Huang, T. Xiang, and X. Liao, "Secure and accurate personalized federated learning with similarity-based model aggregation," *IEEE Trans. Sustain. Comput.*, vol. 10, no. 1, 2025.

[17] J. So, R. E. Ali, B. Güler, and A. S. Avestimehr, "Secure aggregation for buffered asynchronous federated learning," in *Proc. NeurIPS Workshop on New Frontiers in Federated Learning*, 2021.

[18] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated learning with buffered asynchronous aggregation," in *Proc. Artificial Intelligence and Statistics (AISTATS)*, 2022.

[19] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "VerifyNet: Secure and verifiable federated learning," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, 2020.

[20] Y. Zhu, J. Gong, K. Zhang, and H. Qian, "Malicious-resistant non-interactive verifiable aggregation for federated learning," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 6, 2024.

[21] Z. Guan, Y. Zhao, Z. Wan, and W. Wang, "Input integrity and authentic results: Towards trustworthy aggregation in federated learning," in *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, 2025.

[22] K. Garov, D. I. Dimitrov, N. Jovanović, and M. Vechev, "Hiding in plain sight: Disguising data stealing attacks in federated learning," in *Proc. International Conference on Learning Representations (ICLR)*, 2024.

[23] F. Wang and B. Li, "Hear no evil: Detecting gradient leakage by malicious servers in federated learning," *arXiv preprint arXiv:2506.20651*, 2025.

[24] S. Horváth, S. Laskaridis, M. Almeida, I. Leontiadis, S. Venieris, and N. D. Lane, "FjORD: Fair and accurate federated learning under heterogeneous targets with ordered dropout," in *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[25] F. Wu, X. Wang, Y. Wang, T. Liu, L. Su, and J. Gao, "FIARSE: Model-heterogeneous federated learning via importance-aware submodel extraction," in *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[26] "Cloudflare randomness beacon," https://developers.cloudflare.com/randomness-beacon/.

[27] Z. Wang, Z. Chang, J. Hu, X. Pang, J. Du, Y. Chen, and K. Ren, "Breaking secure aggregation: Label leakage from aggregated gradients in federated learning," in *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, 2024.

[28] I. Ergun, H. U. Sami, and B. Guler, "Communication-efficient secure aggregation for federated learning," in *Proc. IEEE Global Communications Conference (GLOBECOM)*, 2022.

[29] R. Lu, Y. Jiang, J. Zhang, C. Li, Y. Zhu, B. Chen, and Z. Wang, "γ-FedHT: Stepsize-aware hard-threshold gradient compression in federated learning," in *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, 2025.

[30] C. Castelluccia, A. C.-F. Chan, E. Mykletun, and G. Tsudik, "Efficient and provably secure aggregation of encrypted data in wireless sensor networks," *ACM Trans. Sens. Netw.*, 2009.

[31] H. Us Sami and B. Güler, "Secure gradient aggregation with sparsification for resource-limited federated learning," *IEEE Trans. Commun.*, vol. 72, no. 11, 2024.

[32] D. Byrd, M. Hybinette, and T. H. Balch, "ABIDES: Towards high-fidelity multi-agent market simulation," in *Proc. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2020.

[33] R. Dorfman, S. Vargaftik, Y. Ben-Itzhak, and K. Y. Levy, "DoCoFL: downlink compression for cross-device federated learning," in *Proc. International Conference on Machine Learning (ICML)*, 2023.

[34] J. Ro, T. Breiner, L. McConnaughey, M. Chen, A. Suresh, S. Kumar, and R. Mathews, "Scaling language model size in cross-device federated learning," in *Proc. ACL Workshop on Federated Learning for Natural Language Processing*, 2022.

[35] N. Strassenburg, I. Tolovski, and T. Rabl, "Efficiently managing deep learning models in a distributed environment," in *Proc. International Conference on Extending Database Technology*, 2022.

[36] R. Lu, Y. Jiang, Y. Mao, C. Tang, B. Chen, L. Cui, and Z. Wang, "Data-aware gradient compression for FL in communication-constrained mobile computing," *IEEE Trans. Mob. Comput.*, vol. 24, no. 4, 2025.

[37] X. Qiu, J. Fernandez-Marques, P. P. B. Gusmao, Y. Gao, T. Parcollet, and N. D. Lane, "ZeroFL: Efficient on-device training for federated learning with local sparsity," in *Proc. International Conference on Learning Representations (ICLR)*, 2022.

[38] A. Guastella, L. Sani, A. Iacob, A. Mora, P. Bellavista, and N. D. Lane, "SparsyFed: Sparse adaptive federated learning," in *Proc. International Conference on Learning Representations (ICLR)*, 2025.