Flamingo: Multi-Round Single-Server Secure Aggregation with Applications to Private Federated Learning

Yiping Ma* Jess Woods* Sebastian Angel*† Antigoni Polychroniadou‡ Tal Rabin*
*University of Pennsylvania †Microsoft Research ‡J.P. Morgan AI Research & AlgoCRYPT CoE

***Abstract*—This paper introduces *Flamingo*, a system for *secure aggregation* of data across a large set of clients. In secure aggregation, a server sums up the private inputs of clients and obtains the result without learning anything about the individual inputs beyond what is implied by the final sum. Flamingo focuses on the multi-round setting found in federated learning in which many consecutive summations (averages) of model weights are performed to derive a good model. Previous protocols, such as Bell et al. (CCS '20), have been designed for a single round and are adapted to the federated learning setting by repeating the protocol multiple times. Flamingo eliminates the need for the per-round setup of previous protocols, and has a new lightweight dropout resilience protocol to ensure that if clients leave in the middle of a sum the server can still obtain a meaningful result. Furthermore, Flamingo introduces a new way to locally choose the so-called client neighborhood introduced by Bell et al. These techniques help Flamingo reduce the number of interactions between clients and the server, resulting in a significant reduction in the end-to-end runtime for a full training session over prior work.**

We implement and evaluate Flamingo and show that it can securely train a neural network on the (Extended) MNIST and CIFAR-100 datasets, and the model converges without a loss in accuracy, compared to a non-private federated learning system.

1. Introduction

In *federated learning*, a server wants to train a model using data owned by many clients (e.g., millions of mobile devices). In each *round* of the training, the server randomly selects a subset of clients, and sends them the current model's weights. Each selected client updates the model's weights by running a prescribed training algorithm on its data locally, and then sends the updated weights to the server. The server updates the model by averaging the collected weights. The training takes multiple such rounds until the model converges.

This distributed training pattern is introduced with the goal of providing a critical privacy guarantee in training—the raw data never leaves the clients' devices. However, prior works [55,77] show that the individual weights still leak information about the raw data, which highlights the need for a mechanism that can securely aggregate the weights computed by client devices [52,74]. This is precisely an instance of *secure aggregation*.

Many protocols and systems for secure aggregation have been proposed, e.g., in the scenarios of private error reporting and statistics collection [5,12,20,24,63,75]. However, secure aggregation in federated learning, due to its specific model, faces unprecedented challenges: a large number of clients, high-dimensional input vectors (e.g., model weights), multiple rounds of aggregation prior to model convergence, and unstable devices (i.e., some devices might go offline prior to completing the protocol). It is therefore difficult to directly apply these protocols in a black-box way and still get good guarantees and performance.

Recent works [9,11,66] propose secure aggregation tailored to federated learning scenarios. In particular, a state-of-the-art protocol [9] (which we call BBGLR) can handle one aggregation with thousands of clients and high-dimensional input vectors, while tolerating devices dropping out at any point during their execution. The drawback of these protocols is that they only examine one round of aggregation in the full training process, i.e., a selection of a subset of the clients and a sum over their inputs.

Utilizing the BBGLR protocol (or its variants) multiple times to do summations in the full training of a model incurs high costs. Specifically, these protocols follow the pattern of having each client establish input-independent secrets with several other clients in a *setup* phase, and then computing a single sum in a *collection* phase using the secrets. These secrets cannot be reused for privacy reasons. Consequently, for each round of aggregation in the training process, one needs to perform an expensive fresh setup. Furthermore, in each *step* (client-server round trip) of the setup and the collection phases, the server has to interact with all of the clients. In the setting of federated learning, such interactions are especially costly as clients may be geographically distributed and may have limited compute and battery power or varying network conditions.

In this paper we propose *Flamingo*, the first single-server secure aggregation system that works well for multiple rounds of aggregation and that can support full sessions of training in the stringent setting of federated learning. At a high level, Flamingo introduces a one time setup and a collection procedure for computing multiple sums, such that the secrets established in the setup can be reused throughout the collection procedure. For each summation in the collection procedure, clients mask their input values (e.g., updated weight vectors in federated learning) with a random mask derived from those secrets, and then send the masked inputs

to the server. The server sums up all the masked inputs and obtains the final aggregate value (what the server wants) masked with the sum of all masks. Flamingo then uses a lightweight protocol whereby a small number of randomly chosen clients (which we call *decryptors*) interact with the server to remove the masks.

The design of Flamingo significantly reduces the overall training time of a federated learning model compared to running BBGLR multiple times. First, Flamingo eliminates the need for a setup phase for each summation, which reduces the number of steps in the full training session. Second, for each summation, Flamingo only has one step that requires the server to contact all of the clients (asking for their inputs); the rest of the interactions are performed between the server and a few clients who serve as decryptors.

Besides training efficiency, the fact that a client needs to only speak once in a round reduces the model's bias towards results that contain only data from powerful, stably connected devices. In Flamingo, the server contacts the clients once to collect inputs; in contrast, prior works have multiple client-server interaction steps in the setup phase (before input collection) and thus filter out weak devices for summation, as staying available longer is challenging. Seen from a different angle, if we fix the number of clients, the failure probability of a client in a given step, and the network conditions, Flamingo's results are of higher quality, as they are computed over more inputs than prior works.

In summary, Flamingo's technical innovations are:

- *Lightweight dropout resilience.* A new mechanism to achieve dropout resilience in which the server only contacts a small number of clients to remove the masks. All other clients are free to leave after the one step in which they submit their inputs without harming the results.
- *Reusable secrets.* A new way to generate masks that allows the reuse of secrets across rounds of aggregation.
- *Per-round graphs.* A new graph generation procedure that allows clients in Flamingo to unilaterally determine (without the help of the server as in prior work) which pairwise masks they should use in any given round.

These advancements translate into significant performance improvements (§8.3). For a 10-round pure summation task, Flamingo is 3× faster than BBGLR (this includes Flamingo's one-time setup cost), and includes the contribution of more clients in its result. When training a neural network on the Extended MNIST dataset, Flamingo takes about 40 minutes to converge while BBGLR needs roughly 3.5 hours to reach the same training accuracy.

2. Problem Statement

Secure aggregation is useful in a variety of domains: collecting, in a privacy-preserving way, error reports [12,24], usage statistics [20,63], and ad conversions [5,75]; it has even been shown to be a key building block for computing private auctions [76]. But one key application is *secure federated learning*, whereby a server wishes to train a model on data that belongs to many clients, but the clients do not wish to share their data (or other intermediate information such as weights that might leak their data [77]). To accomplish this, each client receives the original model from the server and computes new *private* weights based on their own data. The server and the clients then engage in a secure aggregation protocol that helps the server obtain the sum of the clients' private weights, without learning anything about individual weights beyond what is implied by their sum. The server then normalizes the sum to obtain the average weights which represent the new state of the model. The server repeats this process until the training converges.

This process is formalized as follows. Let $[z]$ denote the set of integers $\{1, 2, \ldots, z\}$, and let $\vec{x}$ denote a vector; all operations on vectors are component-wise. A total number of N clients are fixed before the training starts. Each client is indexed by a number in $[N]$. The training process consists of T rounds. In each round $t \in [T]$, a set of clients is randomly sampled from the N clients, denoted as S_t. Each client $i \in S_t$ has an input vector, $\vec{x}_{i,t}$, for round t. (A client may be selected in multiple rounds and may have different inputs in each round.) In each round, the server wants to securely compute the sum of the $|S_t|$ input vectors, $\sum_{i \in S_t} \vec{x}_{i,t}$.

In practical deployments of federated learning, a complete sum is hard to guarantee, as some clients may drop out in the middle of the aggregation process and the server must continue the protocol without waiting for them to come back (otherwise the server might be blocked for an unacceptable amount of time). So the real goal is to compute the sum of the input vectors from the largest possible subset of S_t; we elaborate on this in the next few sections.

2.1. Target deployment scenario

Based on a recent survey of federated learning deployments [43], common parameters are as follows. N is in the range of 100K–10M clients, where $|S_t| = 50$–$5{,}000$ clients are chosen to participate in a given round t. The total number of rounds T for a full training session is 500–10,000. Input weights ($\vec{x}_{i,t}$) have typically on the order of 1K–500K entries for the datasets we surveyed [16,17,46,47].

Clients in these systems are heterogeneous devices with varying degrees of reliability (e.g., cellphones, servers) and can stop responding due to device or network failure.

2.2. Communication model

Each client communicates with the server through a private and authenticated channel. Messages sent from clients to other clients are forwarded via the server, and are end-to-end encrypted and authenticated.

2.3. Failure and threat model

We model failures of two kinds: (1) honest clients that disconnect or are too slow to respond as a result of unstable network conditions, power loss, etc; and (2) arbitrary actions by an adversary that controls the server and a bounded fraction of the clients. We describe each of these below.

Dropouts. In each round of summation, the server interacts with the clients (or a subset of them) several times (several *steps*) to compute a sum. If a server contacts a set of clients during a step and some of the clients are unable to respond in a timely manner, the server has no choice but to keep going; the stragglers are dropped and do *not* participate in the rest of the steps for this summation round. In practice, the fraction of dropouts in the set depends on the client response time distribution and a server timeout (e.g., one second); longer timeouts mean lower fraction of dropouts.

There are two types of clients in the system: regular clients that provide their input, and *decryptors*, who are special clients whose job is to help the server recover the final result. We upper bound the fraction of regular clients that drop out in any given aggregation round by δ, and upper bound the fraction of decryptors that drop out in any given aggregation round by δ_D.

Adversary. We assume a static, malicious adversary that corrupts the server and up to an η fraction of the total N clients. That is, the adversary compromises $N\eta$ clients independent of the protocol execution and the corrupted set stays the same throughout the entire execution (i.e., all rounds). Note that malicious clients can obviously choose to drop out during protocol execution, but to make our security definition and analysis clear, we consider the dropout of malicious clients separate from, and in addition to, the dropout of honest clients.

Similarly to our dropout model, we distinguish between the fraction of corrupted regular clients (η_{S_t}) and corrupted decryptors (η_D). Both depend on η but also on how Flamingo samples regular clients and decryptors from the set of all N clients. We defer the details to Appendix A, but briefly, $\eta_{S_t} \approx \eta$; and given a (statistical) security parameter κ, η_D is upper bounded by $\kappa\eta$ with probability $2^{-\Theta(\kappa)}$.

Threshold requirement. The minimum requirement for Flamingo to work is $\delta_D + \eta_D < 1/3$. For a target security parameter κ, we show in Appendix C how to select other parameters for Flamingo to satisfy the above requirement and result in minimal asymptotic costs.

Comparison to prior works. BBGLR and other works [9, 11] also have a static, malicious adversary but only for a single round of aggregation. In fact, in Section 3.3 we show that their protocol cannot be naturally extended to multiple aggregations that withstands a malicious adversary throughout.

2.4. Properties

Flamingo is a secure aggregation system that achieves the following properties under the above threat and failure model. We give informal definitions here, and defer the formal definitions to Section 5.3.

- *Dropout resilience*: when all parties follow the protocol, the server, in each round t, will get a sum of inputs from all the online clients in S_t. Note that this implicitly assumes that the protocol both completes all the rounds and outputs meaningful results.
- *Security*: for each round t summing over the inputs of clients in S_t, a malicious adversary learns the sum of inputs from at least $(1-\delta-\eta)|S_t|$ clients.

Besides the above, we introduce a new notion that quantifies the quality of a single sum.

- *Sum accuracy*: A round of summation has sum accuracy τ if the final sum result contains the contribution of a τ fraction of the clients who are selected to contribute to that round (i.e., $\tau|S_t|$).

Input correctness. In the context of federated learning, if malicious clients input bogus weights, then the server could derive a bad model (it may even contain "backdoors" that cause the model to misclassify certain inputs [7]). Ensuring correctness against this type of attack is out of the scope of this work; to our knowledge, providing strong guarantees against malicious inputs remains an open problem. Some works [6,10,23,60–62] use zero-knowledge proofs to bound how much a client can bias the final result, but they are unable to formally prove the absence of all possible attacks.

Comparison to prior work. Flamingo provides a stronger security guarantee than BBGLR. In Flamingo, an adversary who controls the server and some clients learns a sum that contains inputs from at least a $1-\delta-\eta$ fraction of clients. In contrast, the malicious protocol in BBGLR leaks several sums: consider a partition of the $|S_t|$ clients, where each partition set has size at least $\alpha \cdot |S_t|$; a malicious adversary in BBGLR learns the sum of each of the partition sets. Concretely, for 5K clients, when both δ and η are 0.2, $\alpha < 0.5$. This means that the adversary learns the sum of two subsets. This follows from Definition 4.1 and Theorem 4.9 in BBGLR [9].

3. Background

In this section, we discuss BBGLR [9], which is the state of the art protocol for a single round secure aggregation in the federated learning setting. We borrow some ideas from this protocol, but design Flamingo quite differently in order to support a full training session.

3.1. Cryptographic building blocks

We start by reviewing some standard cryptographic primitives used by BBGLR and Flamingo.

Pseudorandom generators. A PRG is a deterministic function that takes a random seed in $\{0,1\}^\lambda$ and outputs a longer string that is computationally indistinguishable from a random string (λ is the computational security parameter). For simplicity, whenever we use PRG with a seed that is not in $\{0,1\}^\lambda$, we assume that there is a deterministic function that maps the seed to $\{0,1\}^\lambda$ and preserves security. Such mapping is discussed in detail in Section 6.

Pseudorandom functions. A PRF : $\mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a family of deterministic functions indexed by a key in $\mathcal{K}$ that map an input in $\mathcal{X}$ to an output in $\mathcal{Y}$ in such a way that the

indexed function is computationally indistinguishable from a truly random function from $\mathcal{X}$ to $\mathcal{Y}$. We assume a similar deterministic map for inputs as described in the PRG above.

Shamir's secret sharing. An ℓ-out-of-L secret sharing scheme consists of the following two algorithms, *Share* and *Recon*. $Share(s, \ell, L) \rightarrow (s_1, \ldots, s_L)$ takes in a secret s, a threshold ℓ, and the number of desired shares L, and outputs L shares $s_1, \ldots, s_L$. *Recon* takes in at least $\ell+1$ of the shares, and output the secret s; i.e., for a set $U \subseteq [L]$ and $|U| \geq \ell+1$, $Recon(\{s_u\}_{u \in U}) \rightarrow s$. Security requires that fewer than $\ell+1$ shares reveal no information about s.

Diffie-Hellman key exchange. Let $\mathbb{G}$ be a group of order q in which the Decisional Diffie-Hellman (DDH) problem is hard, and g be a generator of $\mathbb{G}$. Alice and Bob can safely establish a shared secret (assuming a passive adversary) as follows. Alice samples a secret $a \xleftarrow{\$} \mathbb{Z}_q$, and sets her public value to $g^a \in \mathbb{G}$. Bob samples his secret $b \xleftarrow{\$} \mathbb{Z}_q$, and sets his public value to $g^b \in \mathbb{G}$. Alice and Bob exchange the public values and raise the other party's value to their secret, i.e., $g^{ab} = (g^a)^b = (g^b)^a$. If DDH is hard, the shared secret g^{ab} is only known to Alice and Bob but no one else.

3.2. The BBGLR protocol

BBGLR is designed for computing a single sum on the inputs of a set of clients. To apply it to the federated learning setting, we can simply assume that in a given round of the training process, there are n clients selected from a large population of size N. We can then run BBGLR on these n clients to compute a sum of their inputs.

The high level idea of BBGLR is for clients to derive pairwise random masks and to add those masks to their input vectors in such a way that when all the masked input vectors across all clients are added, the masks cancel out. It consists of a *setup* phase and a *collection* phase. We first describe a semi-honest version below.

Setup phase. The setup phase consists of three steps: (1) create a database containing public keys of all of the n clients; (2) create an undirected graph where vertices are clients, and each vertex has enough edges to satisfy certain properties; (3) have each client send shares of two secrets to its neighbors in the graph. We discuss these below.

In the first step, each client $i \in [n]$ generates a secret a_i and sends g^{a_i} to the server, where g^{a_i} represents client i's public key. The server then stores these public keys in a database. Note that the malicious-secure version of BBGLR requires the server to be semi-honest for this particular step, or the existence of a trusted public key infrastructure (PKI).

In the second step, the graph is established as follows. Each client $i \in [n]$ randomly chooses γ other clients in $[n]$ as its neighbors, and tells the server about their choices. After the server collects all the clients' choices, it notifies each client of their neighbors indexes in $[n]$ and public keys. The neighbors of client i, denoted as $A(i)$, are those corresponding to vertices that have an edge with i (i.e., i chose them or they chose i).

Finally, each client i uses Shamir's secret sharing to share a_i and an additional random value m_i to its neighbors $A(i)$ (let the threshold be $\ell < |A(i)|$), where the shares are end-to-end encrypted with a secure authenticated encryption scheme and sent via the server (§2.2).

Collection phase. Client i sends the following masked vector to the server:

$$Vec_i = \vec{x}_i + \underbrace{\sum_{j \in A(i), i<j} \text{PRG}(r_{i,j}) - \sum_{j \in A(i), i>j} \text{PRG}(r_{i,j})}_{\text{pairwise mask}} + \underbrace{\text{PRG}(m_i)}_{\text{individual mask}},$$

where $r_{i,j} = g^{a_i a_j}$, which can be computed by client i since it has the secret a_i and j's public key, g^{a_j}. (These are essentially Diffie-Hellman key exchanges between a client and its neighbors.) Here we view the output of the PRG as a vector of integers instead of a binary string. Also, we will write the pairwise mask term as $\sum_{j \in A(i)} \pm \text{PRG}(r_{i,j})$ for ease of notation.

As we mentioned earlier (§2), clients may drop out due to unstable network conditions, power loss, etc. This means that the server may not receive some of the masked vectors within an acceptable time period. Once the server times out, the server labels the clients whose vectors have been received as "online"; the rest are labeled "offline". The server shares this information with all the n clients. The server then sums up all of the received vectors, which yields a vector that contains the masked sum. To recover the correct sum, the server needs a way to remove the masks. It does so by requesting for each *offline* client i, the shares of a_i from i's neighbors; and for each *online* client j, the shares of m_j from j's neighbors. These shares allow the server to reconstruct either the pairwise mask or the individual mask for each client. As long as there are more than ℓ neighbors that send the requested shares, the server can successfully remove the masks and obtain the sum. This gives the dropout resilience property of BBGLR.

One might wonder the reason for having the individual mask m_i, since the pairwise mask already masks the input. To see the necessity of having m_i, assume that it is not added, i.e., $Vec_i = \vec{x}_i + \sum \pm \text{PRG}(r_{ij})$. Suppose client i's message is sent but not received on time. Thus, the server reconstructs i's pairwise mask $\sum \pm \text{PRG}(r_{ij})$. Then, i's vector Vec_i arrives at the server. The server can then subtract the pairwise mask from Vec_i to learn $\vec{x}_i$. The individual mask m_i prevents this.

Preventing attacks in fault recovery. The above protocol only works in the semi-honest setting. There are two major attacks that a malicious adversary can perform. First, a malicious server can give inconsistent dropout information to honest clients and recover both the pairwise and individual masks. For example, suppose client i has neighbors $j_1, \ldots, j_\gamma$, and a malicious server lies to the neighbors of $j_1, \ldots, j_\gamma$ that $j_1, \ldots, j_\gamma$ have dropped out (when they actually have not). In response, their neighbors, including i, will provide the server with the information it needs to reconstruct $a_{j_1}, \ldots, a_{j_\gamma}$, thereby deriving all the pairwise secrets $r_{i,j_1}, \ldots, r_{i,j_\gamma}$. At the same time, the server can tell $j_1, \ldots, j_\gamma$ that i was online and request the shares of m_i. This gives

the server both the pairwise mask and the individual mask of client i, violating i's privacy. To prevent this, BBGLR has a consistency check step performed among all neighbors of each client to reach an agreement on which nodes actually dropped out. In this case, i would have learned that none of its neighbors dropped out and would have refused to give the shares of their pairwise mask.

Second, malicious clients can submit a share that is different than the share that they received from their neighbors. This could lead to reconstruction failure at the server, or to the server deriving a completely different secret. BBGLR fixes the latter issue by having the clients hash their secrets and send these hashes to the server when they send their input vectors; however, reconstruction could still fail because of an insufficient threshold in error correction[1].

In sum, the full protocol of BBGLR that withstands a malicious adversary (assuming a PKI or a trusted server during setup) has six steps in total: three steps for the setup and three steps for computing the sum.

3.3. Using BBGLR for federated learning

BBGLR works well for *one* round of training, but when many rounds are required, several issues arise. First, in federated learning the set of clients chosen to participate in a round changes, so a new graph needs to be derived and new secrets must be shared. Even if the graph stays the same, the server cannot reuse the secrets from the setup in previous rounds as the masks are in fact one-time pads that cannot be applied again. This means that we must run the setup phase for each round, which incurs a high latency since the setup contains three steps involving all the clients.

Moreover, BBGLR's threat model does not naturally extend to multi-round aggregation. It either needs a semi-honest server or a PKI during the first step of the protocol. If we assume the former, then this means the adversary has to be semi-honest during the exact time of setup in each round, which is practically impossible to guarantee. If we use a PKI, none of the keys can be reused (for the above reasons); as a result, all of the keys in the PKI need to be updated for each round, which is costly.

4. Efficient Multi-Round Secure Aggregation

Flamingo supports multi-round aggregation without redoing the setup for each round and withstands a malicious adversary throughout. The assumptions required are: (1) in the setup, all parties are provided with the same random seed from a trusted source (e.g., a distributed randomness beacon [25]); and (2) a PKI (e.g., a key transparency log [21,41,49,53,68,70,71]). Below we describe the high-level ideas underpinning *Flamingo* (§4.1) and then we give the full protocol (§4.3 and §4.4).

1. To apply a standard error correction algorithm such as Berlekamp-Welch in this setting, the polynomial degree should be at most $\gamma/3$. Definition 4.2 in BBGLR implies that the polynomial degree may be larger than required for error correction.

4.1. High-level ideas

Flamingo has three key ideas:

(1) Lightweight dropout-resilience. Instead of asking clients to secret share a_i and m_i for their masks with all of their neighbors, we let each client encrypt—in a special way—the PRG seeds of their pairwise and individual masks, append the resulting ciphertexts to their masked input vectors, and submit them to the server in a single step. Then, with the help of a special set of L clients that we call *decryptors*, the server can decrypt one of the two seeds associated with each masked input vector, but not both. In effect, this achieves a similar fault-tolerance property as BBGLR (§3.2), but with a different mechanism.

The crucial building block enabling this new protocol is *threshold decryption* [28,32,64], in which clients can encrypt data with a system-wide known public key PK, in such a way that the resulting ciphertexts can only be decrypted with a secret key SK that is secret shared among decryptors in Flamingo. Not only does this mechanism hide the full secret key from every party in the system, but the decryptors can decrypt a ciphertext without ever having to interact with each other. Specifically, the server in Flamingo sends the ciphertext (pealed from the submitted vector) to each of the decryptors, obtains back some threshold ℓ out of L responses, and locally combines the ℓ responses which produce the corresponding plaintext. Our instantiation of threshold decryption is based on the ElGamal cryptosystem and Shamir's secret sharing; we describe it in Section 6. Technically one can also instantiate the threshold decryption with other protocols, but we choose ElGamal cryptosystem because it enables simple distributed key generation and efficient proof of decryption (for robustness, §9).

One key technical challenge that we had to overcome when designing this protocol is figuring out how to secret share the key SK among the decryptors. To our knowledge, existing efficient distributed key generation (DKG) protocols [18,33,57] assume a broadcast channel or reliable point-to-point channels, whereas our communication model is that of a star topology where all messages are proxied by a potential adversary (controlling the server) that can drop them. There are also asynchronous DKG protocols [3,26,44], but standard asynchronous communication model assumes eventual delivery of messages which is not the case in our setting. In fact, we can relax the guarantees of DKG and Section 4.3 gives an extension of a discrete-log-based DKG protocol [33] (since we target ElGamal threshold decryption) that works in the star-topology communication model.

In sum, the above approach gives a dropout-resilient protocol for a single summation with two steps: first, each client sends their masked vector and the ciphertexts of the PRG seeds; second, the server uses distributed decryption to recover the seeds (and the masks) for dropout clients (we discuss how to ensure that decryptors agree on which of the two seeds to decrypt in §4.4). This design improves the run time over BBGLR by eliminating the need to involve all the clients to remove the masks—the server only needs to wait until it has collected enough shares from the decryptors,

instead of waiting for almost all the shares to arrive. Furthermore, the communication overhead of appending several small ciphertexts (64 bytes each) to a large input vector (hundreds of KBs) is minimal.

(2) Reusable secrets. Flamingo's objective is to get rid of the setup phase for each round of aggregation. Before we discuss Flamingo's approach, consider what would happen if we were to naively run the setup phase in BBGLR once, followed by running the collection procedure multiple times. First, we are immediately limited to performing all of the aggregation tasks on the same set of clients, since BBGLR establishes the graph of neighbors during the setup phase. This is problematic since federated learning often chooses different sets of clients for each round of aggregation (§2). Second, clients' inputs are exposed. To see why, suppose that client i drops out in round 1 but not in 2. In round 1, the server reconstructs $r_{i,j}$ for $j \in A(i)$ to unmask the sum. In round 2, client i sends $\vec{x}_i + \sum_{j \in A(i)} \pm \text{PRG}(r_{i,j}) + \text{PRG}(m_i)$ and the server reconstructs m_i by asking i's neighbors for their shares. Since all the $r_{i,j}$ are reconstructed in round 1 and are reused in round 2, the server can derive both masks.

The above example shows that the seeds should be new and independent in each round. We accomplish this with a simple solution that adds a level of indirection. Flamingo treats $r_{i,j}$ as a long-term secret and lets the clients apply a PRF to generate a new seed for each pairwise mask. Specifically, clients i computes the PRG seed for pairwise mask in round t as $h_{i,j,t} := \text{PRF}(r_{i,j}, t)$ for all $j \in A(i)$. Note that client j will compute the same $h_{i,j,t}$ as it agrees with i on $r_{i,j}$. In addition, each client also generates a fresh seed $m_{i,t}$ for the individual mask in round t. Consequently, combined with idea (1), each client uses PK to encrypt the per-round seeds, $\{h_{i,j,t}\}_{j \in A(i)}$ and $m_{i,t}$. Then, the server recovers one of the two for each client. We later describe an optimization where clients do not encrypt $m_{i,t}$ with PK (§4.4).

A nice property of $r_{i,j}$ being a long-term secret is that Flamingo can avoid performing all the Diffie-Hellman key exchanges between graph neighbors (proxied through the server). Flamingo relies instead on an external PKI or a verifiable public key directory such as CONIKS [53] and its successors (which are a common building block for bootstrapping end-to-end encrypted systems).

We note that this simple technique cannot be applied to BBGLR to obtain a secure multi-round protocol. It is possible in Flamingo precisely because clients encrypt their per-round seeds for pairwise masks directly so the server never needs to compute these seeds from the long-term pairwise secrets. In contrast, in BBGLR, clients derive pairwise secrets (g^{a_i,a_j}) during the setup phase. When client i drops out, the server collects enough shares to reconstruct a_i and compute the pairwise secrets, g^{a_i,a_j}, for all online neighbors j of client i. Even if we use a PRF here, the server already has the pairwise secret; so it can run the PRF for any round and conduct the attacks described earlier.

(3) Per-round graphs. BBGLR uses a sparse graph instead of a fully-connected graph for efficiency reasons (otherwise each client would need to secret share its seeds with every other client). In federated learning, however, establishing sparse graphs requires per-round generation since the set S_t changes in each round (some clients are chosen again, some are new [48]). A naive way to address this is to let all clients in $[N]$ establish a big graph G with N nodes in the setup phase: each client in $[N]$ sends its choice of γ neighbors to the server, and the server sends to each client the corresponding neighbors. Then, in each round t, the corresponding subgraph G_t consists of clients in S_t and the edges among clients in S_t.

However, this solution is unsatisfactory. If one uses a small γ (e.g., $\log N$), G_t might not be connected and might even have isolated nodes (leaking a client's input vector since it has no pairwise masks); if one uses a large γ (e.g., the extreme case being N), G_t will not be sparse and the communication cost for the server will be high (e.g., $O(N^2)$).

Flamingo introduces a new approach for establishing the graph with a communication cost independent of γ. The graph for each round t is generated by a random string $v \in \{0,1\}^\lambda$ known to all participants (obtained from a randomness beacon or a trusted setup). Figure 1 lists the procedure. $\text{CHOOSESET}(v, t, n_t, N)$ determines the set of clients involved in round t, namely $S_t \subseteq [N]$ with size n_t. The server computes $G_t \leftarrow \text{GENGRAPH}(v, t, S_t)$ as the graph in round t among clients in S_t. A client $i \in S_t$ can find its neighbors in G_t without materializing the whole graph using $\text{FINDNEIGHBORS}(v, S_t, i)$. In this way, the neighbors of i can be locally generated. We choose a proper ϵ such that in each round, the graph is connected with high probability (details in §5). We note that this technique might be of independent interest.

The above ideas taken together eliminate the need for per-round setup, which improves the overall run time of multi-round aggregation over BBGLR. Figure 2 depicts the overall protocol, and the next sections describe each part.

```
Parameters: ε.                          // the probability that an edge is added
function ChooseSet(v, t, n_t, N)
    S_t ← ∅.
    v*_t := PRF(v, t).
    while |S_t| < n_t do
        Parse log N bits from PRG(v*_t) as i, add i to S_t.
    Output S_t.
function GenGraph(v, t, S_t)
    G_t ← n_t × n_t empty matrix; ρ ← log(1/ε).
    for i ∈ S_t, j ∈ S_t do
        Let v' be the first ρ bits of PRF(v, (i, j)).
        if v' = 0^ρ then set G_t(i, j) := 1
    Output G_t.
function FindNeighbors(v, S_t, i)
    A_t(i) ← ∅; ρ ← log(1/ε).
    for j ∈ S_t do
        Let v' be the first ρ bits of PRF(v, (i, j)).
        if v' = 0^ρ then add j to A_t(i).
    for j ∈ S_t do
        Let v' be the first ρ bits of PRF(v, (j, i)).
        if v' = 0^ρ then add j to A_t(i).
    Output A_t(i).
```

Figure 1: Pseudocode for generating graph G_t in round t.

Figure 2: Workflow of Flamingo. The server first does a setup for all clients in the system. In each round t of training, the server securely aggregates the masked input vectors in the report step; in the cross-check and reconstruction steps, the server communicates with a small set of randomly chosen clients who serve as decryptors. The decryptors are chosen independently from the set S_t that provides inputs in a given round. Every R rounds, the decryptors switch and the old decryptors transfers shares of SK to new decryptors.

4.2. Types of keys at PKI

Before giving our protocol, we need to specify what types of keys the PKI needs to store. The keys depend on the cryptographic primitives that we use (signature schemes, symmetric encryption and ElGamal encryption); for ease of reading, we formally give these primitives in Appendix B.1.

The PKI stores three types of keys for all clients in $[N]$:

- g^{a_i} of client i for its secret a_i; this is for client j to derive the pairwise secret $r_{i,j}$ with client i by computing $(g^{a_j})^{a_i}$.
- g^{b_i} of client i for deriving a symmetric encryption key $k_{i,j}$ for an authenticated encryption scheme SymAuthEnc (Definition 3); this scheme is used when a client sends messages to another client via the server. Later when we say client i sends a message to client j via the server in the protocol, we implicitly assume the messages are encrypted using $k_{i,j}$.
- pk_i of client i for verifying i's signature on messages signed by sk_i.

4.3. Setup phase

The setup phase consists of two parts: (1) distributing a random seed $v \in \{0,1\}^\lambda$ to all participants, and (2) selecting a random subset of clients as decryptors and distribute the shares of the secret key of an asymmetric encryption scheme AsymEnc. In our context, AsymEnc is the ElGamal cryptosystem's encryption function (Definition 2).

As we mentioned earlier, the first part can be done through a trusted source of randomness, or by leveraging a randomness beacon that is already deployed, such as Cloudflare's [1]. The second part can be done by selecting a set of L clients as decryptors, $\mathcal{D}$, using the random seed v (ChooseSet), and then running a DKG protocol among them. We use a discrete-log based DKG protocol [33] (which we call GJKR-DKG) since it is compatible with the ElGamal cryptosystem. However, this DKG does not work under our communication model and requires some changes and relaxations, as we discuss next.

DKG with an untrusted proxy. The correctness and security of the GJKR-DKG protocol relies on a secure broadcast channel. Our communication model does not have such a channel, since the server can tamper, replay or drop messages. Below we give the high-level ideas of how we modify GJKR-DKG and Appendix B.2 gives the full protocol.

We begin with briefly describing the GJKR-DKG protocol. It has a threshold of $1/2$, which means that at most half of the participants can be dishonest; the remaining must perform the following steps correctly: (1) Each party i generates a random value s_i and acts as a dealer to distribute the shares of s_i (party j gets $s_{i,j}$). (2) Each party j verifies the received shares (we defer how the verification is done to Appendix B.2). If the share from the party i fails the verification, j broadcasts a complaint against party i. (3) Party i broadcasts, for each complaint from party j, the $s_{i,j}$ for verification. (4) Each party disqualifies those parties that fail the verification; the rest of the parties form a set QUAL. Then each party sums up the shares from QUAL to derive a share of the secret key.

Given our communication model, it appears hard to guarantee the standard DKG correctness property, which states that if there are enough honest parties, at the end of the protocol the honest parties hold valid shares of a unique secret key. Instead, we relax this correctness property by allowing honest parties to instead abort if the server who is proxying the messages acts maliciously.

We modify GJKR-DKG in the following ways. First, we assume the threshold of dishonest participants is $1/3$. Second, all of the messages are signed; honest parties abort if they do not receive the prescribed messages. Third, we add another step before each client decides on the eventual set QUAL: all parties sign their QUAL set and send it to the server; the server sends the signed QUALs to all the parties. Each party then checks whether it receives $2\ell+1$ or more valid signed QUAL sets that are the same. If so, then the QUAL set defines a secret key; otherwise the party aborts. We give the detailed algorithms and the corresponding proofs in Appendix B.2. Note that the relaxation from GJKR-DKG is that we allow parties to abort (so no secret key is shared at the end), and this is reasonable particularly in the federated learning setting because the server will not get the result if it misbehaves.

Decryptors run DKG. At the end of our DKG, a *subset* of the selected decryptors will hold the shares of the secret key SK. The generated PK is signed by the decryptors and sent to all of the N clients by the server; the signing prevents the server from distributing different public keys or distributing a public key generated from a set of malicious clients. Each client checks if it received $2\ell+1$ valid signed PKs from the set of decryptors determined by the random seed (from beacon); if not, the client aborts. In Appendix B, we provide the pseudocode for the entire setup protocol Π_{setup} (Fig. 11).

4.4. Collection phase

The collection phase consists of T rounds; each round t has three steps: report, cross-check, and reconstruction. Below we describe each step and we defer the full protocol Π_{sum} to Figure 13 in Appendix B. The cryptographic primitives we use here (SymAuthEnc and AsymEnc) are formally given in Appendix B.1.

Report. In round t, the server uses the random value v (obtained from the setup) to select the set of clients $S_t \subseteq [N]$ of size n_t by running $\text{CHOOSESET}(v, t, n_t, N)$. It then establishes the graph $G_t \leftarrow \text{GENGRAPH}(v, t, S_t)$ as specified in Figure 1. We denote the neighbors of i as $A_t(i) \subseteq S_t$. The server asks each client $i \in S_t$ to send a message consisting of the following three things:

1) $Vec_i = \vec{x}_{i,t} + \sum_{j \in A_t(i)} \pm\text{PRG}(h_{i,j,t}) + \text{PRG}(m_{i,t})$, where $h_{i,j,t}$ is computed as $\text{PRF}(r_{i,j}, t)$ and $r_{i,j}$ is derived from the key directory by computing $(g^{a_j})^{a_i}$; $m_{i,t}$ is freshly generated in round t by client i.
2) L symmetric ciphertexts: $\text{SymAuthEnc}(k_{i,u}, m_{i,u,t})$ for all $u \in \mathcal{D}$, where $m_{i,u,t}$ is the share of $m_{i,t}$ meant for u (i.e., $Share(m_{i,t}, \ell, L) \rightarrow \{m_{i,u,t}\}_{u \in \mathcal{D}}$), and $k_{i,u}$ is the symmetric encryption key shared between client i and decryptor u (they can derive $k_{i,u}$ from the PKI);
3) $|A_t(i)|$ ElGamal ciphertexts: $\text{AsymEnc}(PK, h_{i,j,t})$ for all $j \in A_t(i)$.

The above way of using symmetric encryption for individual masks and public-key encryption for pairwise masks is for balancing computation and communication in practice. Technically, one can also encrypt the shares of $h_{i,j,t}$ with symmetric authenticated encryption as well (eliminating public-key operations), but it increases client communication—for each client, the number of ciphertexts appended to the vector is $|A(i)| \cdot L$. This is, for example, 1,600 when L and $|A(i)|$ are both 40. On the other hand, if one encrypts both the pairwise and individual masks using only public-key encryption, then the number of expensive public key operations for reconstructing secrets is proportional to n_t; whereas it is only proportional to the number of dropouts in our proposed approach. In practice, the number of dropouts is much smaller than n_t, hence the savings.

Cross-check. The server needs to recover $m_{i,t}$ for online clients, and $h_{i,j,t}$ for clients that drop out. To do so, the server labels the clients as "offline" or "online" and asks the decryptors to recover the corresponding masks. For BBGLR, we described how this step involves most clients during the fault recovery process and highlighted an issue where a malicious server can send inconsistent labels to clients and recover both the pairwise mask and individual mask for some target client (§3.2). Flamingo also needs to handle this type of attack (the server tells some of honest decryptors to decrypt $m_{i,t}$ and other honest decryptors to decrypt $h_{i,j,t}$, and utilizes the malicious decryptors to reconstruct both), but it only needs to involve decryptors. In detail, each decryptor signs the online/offline labels of the n_t clients (each client can only be labeled either offline or online), and sends them to the other decryptors (via the server). Each decryptor checks it received $2L/3$ or more valid signed labels (recall from §2.3 that $\delta_D + \eta_D < 1/3$). If so, each decryptor further checks that:

1) The number of online clients is at least $(1-\delta)n_t$;
2) All the online clients in the graph are connected;
3) Each online client i has at least k online neighbors, such that $\eta^k < 2^{-\kappa}$ (η and κ are defined as in §2.3).

If any of the above checks fail, the decryptor aborts. This step ensures either all the honest decryptors agree on a valid offline/online label assignment and consequently the server gets the result, or the honest decryptors abort and the server gets nothing.

Reconstruction. The server collects all the ciphertexts to be decrypted: the ciphertexts of $m_{i,u,t}$ (symmetric encryption) for the online clients, and the ciphertexts of $h_{i,j,t}$ (public-key encryption) for the offline clients. Then the server sends the ciphertexts to all the decryptors who perform either a symmetric decryption or the threshold ElGamal decryption according to their agreed-upon labels.

The choice of using decryptors to check the graph and reconstruct all the secrets is based on an important observation in federated learning: the number of clients involved in one round, n_t, is much smaller than the input vector length [43]. Therefore, the asymptotic costs at a decryptor (which are proportional to n_t) are actually smaller than the size of an input weight vector.

4.5. Malicious labeling across rounds

The server, controlled by a malicious adversary, can ask for the decryption of $h_{i,j,t}$ in round t, and then in some other round t', the server can ask for the decryption of $m_{i,t}$ (but not $m_{i,t'}$, if the server does not care about obtaining a result in round t'). This allows the server to recover $\vec{x}_{i,t}$ in the clear. To prevent this attack, honest decryptors need to know the round for which a ciphertext is sent. For symmetric ciphertext, the client appends the round number t to the plaintext (e.g., $m_{i,u,t}||t$) and uses authenticated encryption; for public-key ciphertexts, the client appends t to the ciphertext c and signs the tuple (c, t) (the verification key is in the PKI). Note that a malicious adversary can still fool enough honest decryptors into thinking it is round t while it is in fact t'. To prevent this, decryptors also include the round number in the online/offline labels and sign them. The cross-check (§4.4) guarantees that the decryptors agree on the round number.

4.6. Load balancing across decryptors

In each summation, a client who is not a decryptor only sends a single vector. This is nearly optimal since even if the aggregation is non-private the client has to send the vector (but without the additional small ciphertexts). The decryptors, however, have additional computation and communication in order to help with the result reconstruction. This causes a load imbalance in the system and could be unfair since a client selected to be a decryptor has to do more work than regular clients.

In Flamingo, the decryptor responsibility shifts across time. Every R rounds, the current decryptors transfer their shares of SK to a new set of randomly selected clients who serve as the new decryptors. To ensure security, the shares of SK have to be modified in a particular way during the transition, as otherwise the adversary may control some malicious decryptors before the transition and some malicious decryptors after the transition, and thus may obtain enough shares to reconstruct SK. We address this by relying on prior proactive secret sharing techniques [32,40,45]; they additionally enable Flamingo to change the number of decryptors and the threshold as needed. In Appendix B.4, we provide details of the transition protocol used in Flamingo.

A final clarification is that decryptors who dropped out (e.g., due to power loss) at one round can come back later and participate in another round (e.g., when power is resumed). The decryption always succeeds since we require that less than $1/3$ decryptors are dropped out or malicious at any step (§5). The secret key transition is purely for system load balancing—neither dropout resilience nor security relies on the parameter R.

4.7. Considerations in federated learning

A recent work [56] describes an attack in the composition of federated learning and secure aggregation. The idea is that the server can elude secure aggregation by sending clients inconsistent models. For example, the server sends to client 1 model M_1, to client 2 model M_2, and to client 3 model M_3. Each of the clients then runs the local training on the provided model. The server chooses the models it sends to clients 1 and 2 in a special way such that after clients 1 and 2 train their local models, their local weights will cancel out when added. Consequently, the server will get the model weights of client 3. A potential defense, which works in our context without incurring any overhead, is for clients to append the hash of the model they receive in a given round to their PRG seed for that round: $\mathrm{PRG}(h_{i,j,t}||Hash(M))$, where M is the model received from the server. If all the clients receive the same model, the pairwise masks cancel out; otherwise, they do not.

5. Parameter Selection and Security Analysis

The parameters of Flamingo include:

- System parameters N, T and the number of clients n_t chosen in round $t \in [T]$;
- Threat model parameters δ_D, δ, η which are given, and η_{S_t}, η_D which depend on η (their relation is summarized in Section 2.3 and fully derived in Appendix A).
- Security parameter κ, and the parameters that relates to security: graph generation parameter ϵ, and the number of selected decryptors L.

We discuss these parameters in detail below and state our formal lemmas with respect to them.

Functionality $\mathcal{F}_{\text{setup}}$

Parties: clients $1, \ldots, N$ and a server.

- $\mathcal{F}_{\text{setup}}$ samples $v \xleftarrow{\$} \{0,1\}^\lambda$.
- $\mathcal{F}_{\text{setup}}$ samples a secret key and public key pair (SK, PK).
  // When the public-key cryptosystem is instantiated by ElGamal, then SK is $s \xleftarrow{\$} \mathbb{Z}_q$ and $PK = g^s$.
- $\mathcal{F}_{\text{setup}}$ asks the adversary $\mathcal{A}$ whether it should continue or not. If $\mathcal{A}$ replies with `abort`, $\mathcal{F}_{\text{setup}}$ sends `abort` to all honest parties; if $\mathcal{A}$ replies with `continue`, $\mathcal{F}_{\text{setup}}$ sends v and PK to all the parties.

Figure 3: Ideal functionality for the setup phase.

5.1. Security of setup phase

Let δ_D upper bound the fraction of decryptors that drop out during the setup phase; note that in Section 2.3 we let δ_D upper bound the dropouts in one aggregation round and for simplicity here we use the same notation. Flamingo's DKG requires that $\delta_D + \eta_D < 1/3$. Note that η_D in fact depends on η, L and N, but we will give the theorems using η_D and discuss how to choose L to guarantee a desired η_D in Appendix C.

Theorem 1 (Security of setup phase)**.** Assume that a PKI and a trusted source of randomness exist, and that the DDH assumption holds. Let the dropout rate of decryptors in the setup phase be bounded by δ_D. If $\delta_D + \eta_D < 1/3$, then under the communication model defined in Section 2.2, protocol Π_{setup} (Fig. 11) securely realizes functionality $\mathcal{F}_{\text{setup}}$ (Fig. 3) in the presence of a malicious adversary controlling the server and η fraction of the N clients.

5.2. Security of collection phase

First, we need to guarantee that each graph G_t, even *after* removing the vertices corresponding to the $\delta + \eta$ fraction of dropout and malicious clients, is still connected. This imposes a requirement on ϵ, which we state in Lemma 1. For simplicity, we omit the exact formula for the lower bound of ϵ and defer the details to Appendix C.

Lemma 1 (Graph connectivity)**.** Given a security parameter κ, and threat model parameters δ, η (§2.3). Let G be a random graph $G(n, \epsilon)$. Let $\mathcal{C}, \mathcal{O}$ be two random subsets of nodes in G where $|\mathcal{O}| \leq \delta n$ and $|\mathcal{C}| \leq \eta n$ ($\mathcal{O}$ stands for dropout set and $\mathcal{C}$ stands for malicious set). Let $\tilde{G}$ be the graph with nodes in $\mathcal{C}$ and $\mathcal{O}$ and the associated edges removed from G. There exists ϵ^* such that for all $\epsilon \geq \epsilon^*$, $\tilde{G}$ is connected except with probability $2^{-\kappa}$.

Secondly, we require $2\delta_D + \eta_D < 1/3$ to ensure that all online honest decryptors reach an agreement in the cross-check step and the reconstruction is successful. Note that the decryptors in the setup phase who dropped out (δ_D fraction) will not have the share of SK; while the clients who drop

Functionality $\mathcal{F}_{\text{mal}}$

Parties: clients $1, \ldots, N$ and a server.

Parameters: corrupted rate η, dropout rate δ.

- $\mathcal{F}_{\text{mal}}$ receives from the adversary $\mathcal{A}$ a set of corrupted parties, denoted as $\mathcal{C} \subset [N]$, where $|\mathcal{C}|/N \leq \eta$.
- For each round $t \in [T]$:
  1) $\mathcal{F}_{\text{mal}}$ receives a random subset $S_t \subset [N]$, a dropout set $\mathcal{O}_t \subset S_t$, where $|\mathcal{O}_t|/|S_t| \leq \delta$, and inputs $\vec{x}_{i,t}$ of client $i \in S_t \backslash (\mathcal{O}_t \cup \mathcal{C})$.
  2) $\mathcal{F}_{\text{mal}}$ sends S_t to $\mathcal{A}$ and asks $\mathcal{A}$ for a set and whether it should continue or not: if $\mathcal{A}$ replies with `continue` and a set $M_t \subseteq S_t \backslash \mathcal{O}_t$ such that $|M_t|/|S_t| \geq 1-\delta$, then $\mathcal{F}_{\text{mal}}$ outputs $z'_t = \sum_{i \in M_t \backslash \mathcal{C}} \vec{x}_{i,t}$; otherwise sends `abort` to all honest clients in S_t.

Figure 4: Ideal functionality for Flamingo.

out during a round (another δ_D fraction) in the collection phase can come back at another round, hence we have the above inequality.

5.3. Main theorems

The full protocol, denoted as Φ_T, is the sequential execution of Π_{setup} (Fig. 11) followed by a T-round Π_{sum} (Fig. 13). We now give formal statements for the properties of Flamingo, and defer the proof to Appendix D. Note that as we see from the ideal functionality $\mathcal{F}_{\text{mal}}$ (Fig. 4), when the server is corrupted, the sum result in round t is not determined by the actual dropout set $\mathcal{O}_t$, but instead a set M_t chosen by the adversary (see details in Appendix D.5).

Theorem 2 (Dropout resilience of Φ_T)**.** Let the security parameter be κ. Let $\delta, \delta_D, \eta, \eta_D$ be threat model parameters as defined (§5.1,§5.2). If $2\delta_D + \eta_D < 1/3$, then protocol Φ_T satisfies dropout resilience: when all parties follow the protocol Φ_T, for every round $t \in [T]$, and given a set of dropout clients $\mathcal{O}_t$ in the report step where $|\mathcal{O}_t|/|S_t| \leq \delta$, protocol Φ_T terminates and outputs $\sum_{i \in S_t \backslash \mathcal{O}} \vec{x}_{i,t}$, except probability $2^{-\kappa}$.

Theorem 3 (Security of Φ_T)**.** Let the security parameter be κ. Let $\delta, \delta_D, \eta, \eta_D$ be threat model parameters as defined (§5.1,§5.2). Let ϵ be the graph generation parameter (Fig. 1). Let n be the number of clients for summation in each round. Assuming the existence of a PKI, a trusted source of initial randomness, a PRG, a PRF, an asymmetric encryption AsymEnc, a symmetric authenticated encryption SymAuthEnc, and a signature scheme, if $2\delta_D + \eta_D < 1/3$ and $\epsilon \geq \epsilon^*$ (Lemma 1) and , then under the communication model defined in Section 2.2, protocol Φ_T securely realizes the ideal functionality $\mathcal{F}_{\text{mal}}$ given in Figure 4 in the presence of a malicious adversary controlling the server and η fraction of the N clients, except with probability at most $Tn \cdot 2^{-\kappa+1}$.

Additionally, if the asymmetric encryption is instantiated by ElGamal cryptosystem, the security can be based on the DDH assumption and the other primitives stated above.

The final complication is how to choose L to ensure $2\delta_D + \eta_D < 1/3$ holds; note that η_D depends on η and L. One can choose L to be N but it does not give an efficient protocol; on the other hand, choosing a small L may result in all the decryptors being malicious. In Appendix C, we give a lower bound of L to ensure a desired η_D (w.h.p.), given N, η, and δ_D.

6. Implementation

We implement Flamingo in 1.7K lines and BBGLR in 1.1K lines of Python. For PRG, we use AES in counter mode, for authenticated encryption we use AES-GCM, and signatures use ECDSA over curve P-256. Our code is available at `https://github.com/eniac/flamingo`.

Distributed decryption. We build the distributed decryption scheme discussed in Section 4.1 as follows. We use ElGamal encryption to instantiate the asymmetric encryption. It consists of three algorithms (AsymGen, AsymEnc, AsymDec). AsymGen outputs a secret and public key pair $SK \in_R \mathbb{Z}_q$ and $PK := g^{SK} \in \mathbb{G}$. AsymEnc takes in PK and plaintext $h \in \mathbb{G}$, and outputs ciphertext $(c_0, c_1) := (g^w, h \cdot PK^w)$, where $w \in_R \mathbb{Z}_q$ is the encryption randomness. AsymDec takes in SK and ciphertext (c_0, c_1) and outputs $h = (c_0^{SK})^{-1} \cdot c_1$.

In threshold decryption [28,32,64], the secret key SK is shared among L parties such that each party $u \in [L]$ holds a share s_u, but no single entity knows SK, i.e., $(s_1, \ldots, s_L) \leftarrow Share(SK, \ell, L)$. Suppose Alice wants to decrypt the ciphertext (c_0, c_1) using the secret-shared SK. To do so, Alice sends c_0 to each party in $[L]$, and gets back $c_0^{s_u}$ for $u \in U \subseteq [L]$. If $|U| > \ell$, Alice can compute from U a set of combination coefficients $\{\beta_u\}_{u \in U}$, and

$$c_0^{SK} = \prod_{u \in U} (c_0^{s_u})^{\beta_u}.$$

Given c_0^{SK}, Alice can get the plaintext $h = (c_0^{SK})^{-1} \cdot c_1$. Three crucial aspects of this protocol are that: (1) SK is never reconstructed; (2) the decryption is dropout resilient (Alice can obtain h as long as more than ℓ parties respond); (3) it is non-interactive: Alice communicates with each party exactly once.

We implement ElGamal over elliptic curve group $\mathbb{G}$ and we use curve P-256. To map the output of $\text{PRF}(r_{i,j}, t)$, which is a binary string, to $\mathbb{G}$, we first hash it to an element in the field of the curve using the *hash-to-field* algorithm from the IETF draft [39, §5]. We then use the SSWU algorithm [13, 72] to map the field element to a curve point $P \in \mathbb{G}$. A client will encrypt P with ElGamal, and then hash P with SHA256 to obtain $h_{i,j,t}$—the input to the pairwise mask's PRG. When the server decrypts the ElGamal ciphertexts and obtains P, it uses SHA256 on P to obtain the same value of $h_{i,j,t}$.

Optimizations. In Flamingo's reconstruction step, we let the server do reconstruction using partial shares. That is, if the interpolation threshold is 15, and the server collected shares from 20 decryptors, it will only use 15 of them and ignore the remaining 5 shares. Furthermore, as we only have

		BBGLR			Flamingo		
Phase		**Steps**	**Server**	**Client**	**Steps**	**Server**	**Client**
Setup		—	—	—	4	$O(L^3)$	$O(L^2)$
T sums	Round setup	$3T$	$O(TAn_t)$	$O(TA)$	—	—	—
	Collection	$3T$	$O(Tn_t(d+A))$	$O(T(d+A))$	$3T$	$O(Tn_t(d+L+A))$	Regular client: $O(T(d+A))$ Decryptors: $O(T(L+\delta An_t+(1-\delta)n_t))$

Figure 5: Communication complexity and number of steps (client-server round-trips) of Flamingo and BBGLR for T rounds of aggregation. N is the total number of clients and n_t is the number of clients chosen to participate in round t. The number of decryptors is L, and the dropout rate of clients in S_t is δ. Let A be the upper bound on the number of neighbors of a client, and let d be the dimension of client's input vector.

a single set of decryptors, when the server collects shares from $U \subseteq \mathcal{D}$, it computes a single set of interpolation coefficients from U and uses it to do linear combinations on all the shares. This linear combination of all the shares can be done in parallel. In contrast, BBGLR requires the server to compute different sets of interpolation coefficients to reconstruct the pairwise masks (one set for each client).

Simulation framework. We integrade all of Flamingo's code into ABIDES [14,15], which is an open-source high-fidelity simulator designed for AI research in financial markets (e.g., stock exchanges). ABIDES is a great fit as it supports tens of thousands of clients interacting with a server to facilitate transactions (and in our case to compute sums). It also supports configurable pairwise network latencies.

7. Asymptotic Costs

An advantage of Flamingo over BBGLR is that Flamingo requires fewer round trips for one summation. BBGLR requires six round trips for one summation; in contrast, Flamingo requires three: the report step involves all clients selected in a round, and for the remaining two steps the server contacts the decryptors. Meanwhile, the number of round trips has a significant impact on the overall runtime of the aggregation, as we show experimentally in Section 8.2. The reasons for this are two-fold: (1) the latency of an RTT over the wide area network is on the order of tens of milliseconds; and (2) the server has to wait for enough clients in each step to send their messages, so tail latency plays a big role. Depending on the setting, client message arrival can vary widely, with some clients potentially being mobile devices on slow networks.

In addition to fewer round trips, Flamingo is expected to wait for less time during the reconstruction step. This is because the server in BBGLR has to wait for the vast majority of clients to respond in order to reconstruct secrets for dropout clients, while the server in Flamingo only needs responses from $1/3$ of the decryptors.

Client and server costs. Figure 5 compares Flamingo's communication costs with those of BBGLR. In short, the total asymptotic cost for T aggregation rounds between BBGLR and Flamingo does not vary much; but the number of round-trips differ much. The computation cost is analyzed as follows. For the setup phase, if a client is a decryptor, it has $O(L^2)$ computation for both DKG, and secret key transfer. In the collection phase at round t, each client computes $O(A+L)$ encryptions (assuming that for all clients i, $A(i) \leq A$). If a client is a decryptor, it additionally has a $O(\delta An_t + (1-\delta)n_t + \epsilon n_t^2)$ computation cost.

8. Experimental Evaluation

In this section we answer the following questions:

- What are Flamingo's concrete server and client costs, and how long does it take Flamingo to complete one and multiple rounds of aggregation?
- Can Flamingo train a realistic neural network?
- How does Flamingo compare to the state of the art in terms of the quality of the results and running time?

We implement the following baselines:

Non-private baseline. We implement a server that simply sums up the inputs it receives from clients. During a particular round, each of the clients sends a vector to the server. These vectors are in the clear, and may be any sort of value (e.g. floating points), unlike Flamingo, which requires data to be masked positive integers. The server tolerates dropouts, as Flamingo does, and aggregates only the vectors from clients who respond before the timeout.

BBGLR. For BGGLR, we implement Algorithm 3 in their paper [9] with a slight variation that significantly improves BBGLR's running time, although that might introduce security issues (i.e., we are making this baseline's performance better than it is in reality, even if it means it is no longer secure). Our specific change is that we allow clients to drop out during the graph establishment procedure and simply construct a graph with the clients that respond in time. BBGLR (originally) requires that no client drops out during graph establishment to ensure that the graph is connected. Without this modification, BBGLR's server has to wait for all the clients to respond and is severely impacted by the long tail of the client response distribution—which makes our experiments take an unreasonable amount of time.

8.1. Experimental environment

Prior works focus their evaluation on the server's costs. While this is an important aspect (and we also evaluate it), a key contributor to the end-to-end completion time of the aggregation (and of the federated learning training) is the number of round-trips between clients and the server. This is especially true for geodistributed clients.

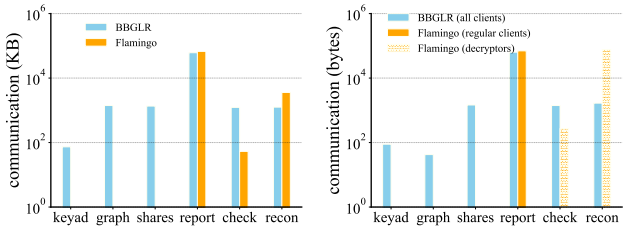
To faithfully evaluate real network conditions, we run the ABIDES simulator [14] on a server with 40 Intel Xeon E5-2660 v3 (2.60GHz) CPUs and 200 GB DDR4 memory. Note that in practice, client devices are usually less powerful than the experiment machine. ABIDES supports the cubic network delay model [37]: the latency consists of a base delay (a range), plus a jitter that controls the percentage of messages that arrive within a given time (i.e., the shape of the delay distribution tail). We set the base delay to the "global" setting in ABIDES's default parameters (the range is 21 microseconds to 53 milliseconds), and use the default parameters for the jitter.

Both Flamingo and BBGLR work in steps (as defined in §2.3). We define a *waiting* period for each step of the protocol. During the waiting period, the server receives messages from clients and puts the received messages in a message pool. When the waiting period is over, a *timeout* is triggered and the server processes the messages in the pool, and proceeds to the next step. The reason that we do not let the server send and receive messages at the same time is that, in some steps (in both BBGLR and Flamingo), the results sent to the clients depend on all the received messages and cannot be processed in a streaming fashion. For example, the server must decide on the set of online clients before sending the request to reconstruct the shares.

8.2. Secure aggregation costs and completion time

This section provides microbenchmarks for summation tasks performed by BBGLR and Flamingo. Client inputs are 16K-dimensional vectors with 32-bit entries. For parameters, unless specified differently later, we set N to 10K and the number of decryptors to 60 in Flamingo; and set the number of neighbors to $4 \log n_t$ for both Flamingo and BBGLR (for BBGLR, this choice satisfies the constraints in Lemma 4.7 [9]). In Figures 6 and 7, "keyad" is the step for exchanging keys, "graph" is the step for clients to send their choices of neighbors, "share" is the step for clients to shares their secrets to their neighbors (marked as 1–8 in their Algorithm 3). The steps "report", "check" and "recon" in Flamingo are described in Section 4.4; in BBGLR, these steps correspond to the last three round trips in a summation marked as 8–14 in their Algorithm 3.

Communication costs. Figure 6 gives the communication cost for a single summation. The total cost per aggregation round for BBGLR and Flamingo are similar. This is because Flamingo's extra cost over BBGLR at the report step is roughly the message size that BBGLR has in their three-step setup; this is also reflected in the asymptotic cost analysis of Figure 5. In short, compared to BBGLR, Flamingo has fewer round trips with roughly the same total server communication. For clients, the story is more nuanced: each client has a slightly higher cost in Flamingo than in BBGLR during the report step, as clients in Flamingo need to append ciphertexts to the vector. However, in the reconstruction step, clients who are not decryptors will not need to send or receive any messages. Each decryptor incurs communication that is slightly larger than sending one input vector. Note that the vector size only affects the report step.



(a) Server communication (b) Client communication

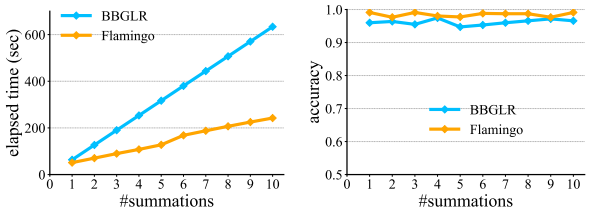
Figure 6: Communication costs for different steps in a single summation over 1K clients for Flamingo and BBGLR.

CPU costs	keyad	graph	share	report	check	recon
Server (sec)						
BBGLR	0.11	0.27	0.09	0.09	0.08	0.76
Flamingo	—	—	—	0.24	—	2.30
Client (sec)						
BBGLR	<0.01	<0.01	<0.01	0.02	0.01	<0.01
Flamingo						
Regular clients	—	—	—	0.22	—	—
Decryptors	—	—	—	—	0.10	0.56

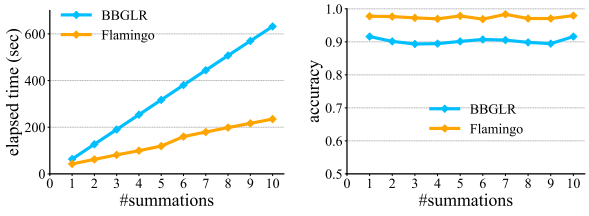
Figure 7: Single-threaded microbenchmarks averaged over 10 runs for server and client computation for a single summation over 1K clients. "<" means less than.

Computation costs. We first microbenchmark a single summation with 1K clients, and set δ to 1% (i.e., up to 1% of clients can drop out in any given step). This value of δ results in a server waiting time of 10 seconds. Figure 7 gives the results. The report step in Flamingo has slightly larger server and client costs than BBGLR because clients need to generate the graph "on-the-fly". In BBGLR, the graph is already established in the first three steps and stored for the report and reconstruction step. For server reconstruction time, Flamingo is slightly more costly than BBGLR because of the additional elliptic curve operations. The main takeaway is that while Flamingo's computational costs are slightly higher than BBGLR, these additional costs have negligible impact on completion time owing to the much larger effect of network delay, as we show next.

Aggregation completion time. To showcase how waiting time w affects dropouts (which consequently affects the sum accuracy), we use two waiting times, 5 seconds and 10 seconds. The runtime for an aggregation round depends on the timeout for each step, the simulated network delay, and the server and client computation time. Figure 8a and 8c



(a) Runtime with $w = 10$. (b) Sum accuracy τ; $w = 10$.



(c) Runtime with $w = 5$. (d) Sum accuracy τ; $w = 5$.

Figure 8: End-to-end completion time and accuracy of 10 secure aggregation rounds with 1K clients. The elapsed time is the finishing time of round t. For Flamingo, round 1 includes all of the costs of its one-time setup, and between round 5 and 6 Flamingo performs a secret key transfer.

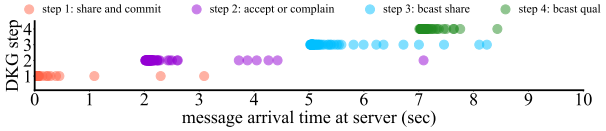
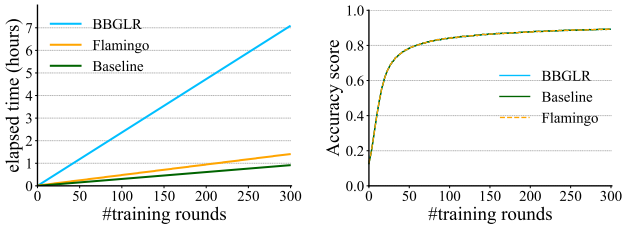


Figure 9: Generating shares of the secret key among 60 decryptors. The four steps are described in Section 4 and given as part (1) in Π_{DKG} in Appendix B.2.

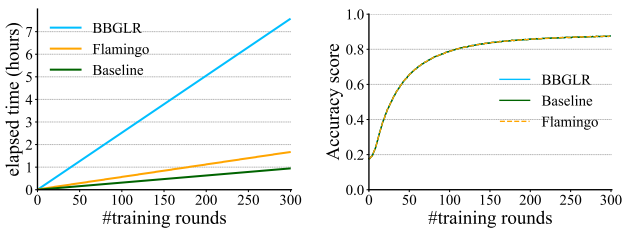
show the overall completion time across 10 rounds of aggregations. A shorter waiting time makes the training faster, but it also means that there are more dropouts, which in turn leads to more computation during reconstruction. As a result, the overall runtime for the two cases are similar. On 1K clients, Flamingo achieves a 3× improvement over BBGLR; for Flamingo's cost we included its one-time setup and one secret key transfer. If the key transfer is performed less frequently, the improvement will be more significant.

The cost of the DKG procedure (part of the setup and which we also added to the first round in Figure 8) is shown in Figure 9. A complete DKG takes less than 10 seconds as the number of decryptors is not large and we allow decryptor dropouts. For 60 decryptors, the local computation performed by each decryptor during the DKG is 2 seconds.

Summation accuracy. Figure 8b and 8d show that Flamingo achieves better sum accuracy τ (defined in §2.4) than BBGLR when they start a summation with the same number of clients. When the waiting time is shorter, as in Figure 8d, in each step there are more clients excluded from the summation and therefore the discrepancy between Flamingo and BBGLR grows larger.



(a) Run time. `EMNIST` (b) Accuracy. `EMNIST`



(c) Run time. `CIFAR100` (d) Accuracy. `CIFAR100`

Figure 10: Evaluation for full training sessions on `EMNIST` and `CIFAR100` datasets. The number of clients per round is 128, the batch and epoch sizes for `FedAvg` are 10 and 20, respectively. Flamingo's setup cost is included during the first round, and it performs a secret key transfer every 20 rounds, which adds to the total run time. The accuracy score is TensorFlow's sparse categorical accuracy score [2].

8.3. Feasibility of a full private training session

We implement the federated learning algorithm `FedAvg` [52] on the non-private baseline. We also use this algorithm for Flamingo and BBGLR but replace its aggregation step with either Flamingo or BBGLR to produce a secure version. Inside of `FedAvg`, we use a multilayer perceptron for image classification. Computation of the weights is done separately by each client on local data, and then aggregated by the server to update a global model. The server then sends the global model back to the clients. The number of training iterations that clients perform on their local data is referred to as an *epoch*. We evaluated Flamingo on epochs of size 5, 10, and 20. We found that often, a larger epoch was correlated with faster convergence of `FedAvg` to some "maximum" accuracy score. Additionally, because our neural network model calculations run very fast—there was, on average, less than a second difference between clients' model fitting times for different epochs—and because Flamingo and the baselines were willing to wait for clients' inputs for up to 10 seconds, the epoch size did not affect their overall runtime.

We use two of TensorFlow's federated datasets [2]: (1) `EMNIST`, the Extended MNIST letter dataset from the Leaf repository [16,17]; and (2) `CIFAR100`, from the CIFAR-100 tiny images dataset [46,47]. The `EMNIST` dataset has ∼340K training/∼40K test samples, each with a square of 28×28 pixels and 10 classes (digits). Each client has ∼226 samples. During training, we use weight vectors with 8K 32-bit entries. The `CIFAR100` dataset has 50K training/10K

test samples, each with a square of 32×32 pixels and 100 classes. Each pixel additionally has red/blue/green values. Each client has 100 samples. To achieve good accuracy for the `CIFAR100` dataset, we use a more complex convolutional neural network than we do for the `EMNIST` dataset, with extra layers to build the model, normalize inputs between layers, and handle activation functions and overfitting. This results in longer weight vectors, with 500K 32-bit entries.

We randomly divide the datasets equally among 128 clients to create local data. Local models are trained with small batch sizes. In Flamingo and BBGLR, all weights (often floating point numbers) are encoded as positive integers. We do this by adding a large positive constant, multiplying by 2^{12}, and truncating the weight to an unsigned 32-bit integer. Figure 10 shows the result with $\delta = 1\%$.

Running time. From Figures 10b and 10d, we see that the `EMNIST` and `CIFAR100` datasets do not converge until about round 150 and 200, respectively, though their accuracy continues to improve slightly after that. Figures 10a and 10c show Flamingo's running time is about 5.5× lower (i.e., better) than BBGLR for `EMNIST` and 4.8× for `CIFAR100` and about 1.4× higher (i.e., worse) than the non-private baseline for `EMNIST` and 1.7× for `CIFAR100`. We believe these results provide evidence that Flamingo is an effective secure aggregation protocol for multi-round settings such as those required in federated learning.

Training accuracy. We measure training accuracy with TensorFlow's sparse categorical accuracy score, which is derived based on the resulting model's performance on test data. Due to the way in which we encode floating points as integers, a small amount of precision from the weights is lost each round. We compare the accuracy of Flamingo and BBGLR's final global model to a model trained on the same datasets with the baseline version of `FedAvg` (which works over floating points) in Figures 10b and 10d. We find that the encoding itself does not measurably affect accuracy.

9. Extension with robustness

Recall that our threat model (§2.3) assumes that the server is controlled by the adversary. However, in some cases, the server is actually honest and wants to obtain a meaningful result—this motivates the need of having the protocol be *robust* against malicious clients in addition to the other security properties. In Section 3.2, we discussed that in BBGLR, when the server is honest, a malicious client who deviates from the protocol may cause the output to be wrong or the protocol to abort (without being detected). The attack is simple: a malicious client changes the received share of a secret, and if the server cannot reconstruct the secret then the protocol aborts; or the server may reconstruct to another secret. The protocol we give in Section 4 does not have robustness for the same reason as in BBGLR, but we provide an extension such that the malicious behavior of the clients can be detected. The full description of the extension is given in Appendix E.1.

As a result, the extended protocol always terminates with the correct output (i.e., the sum of the inputs from clients who participated) when the server is honest; if the server is malicious, then the protocol either aborts or outputs a sum from at least $(1 - \delta - \eta)|S_t|$ clients. That said, we want to emphasize that this difference in security is not practically meaningful at the moment since malicious clients are free to provide any input they want (see the paragraph on input correctness in §2.4). However, it could become important in the future: if there is ever some mechanism that could confirm the validity of clients' inputs, then this additional guarantee would ensure that an honest server always gets valid outputs even if malicious clients exist in the system.

10. Related Work

In this section we discuss alternative approaches to compute private sums and the reasons why they do not fit well in the setting of federated learning. Readers may also be interested in a recent survey of this area [51].

Pairwise masking. Bonawitz et al. [11] and follow up works [9,65], of which BBGLR [9] is the state-of-the-art, adopt the idea of DC networks [22] in which pairwise masks are used to hide individuals' inputs. Such a construction is critical for both client-side and server-side efficiency: first, since the vectors are long, one-time pad is the most efficient way to encrypt a vector; second, the server just needs to add up the vectors, which achieves optimal server computation (even without privacy, the server at least has to do a similar sum). Furthermore, pairwise masking protocols support flexible input vectors, i.e., one can choose any b (the number of bits for each component in the vector) as desired. Flamingo improves on this line of work by reducing the overall round trip complexity for multiple sums.

MPC. Works like FastSecAgg [42] use a secret-sharing based MPC to compute sums, which tolerates dropouts, but it has high communication as the inputs in federated learning are large. Other results use non-interactive MPC protocols for addition [65,66] where all the clients establish shares of zero during the setup. And then when the vectors of clients are requested, each client uses the share to mask the vector and sends it to the server. However, to mask long vectors, the clients need to establish many shares of zeros, which is communication-expensive. Such shares cannot be reused over multiple summation rounds (which is precisely what we address with Flamingo). Furthermore, the non-interactive protocols are not resilient against even one dropout client.

Additively homomorphic encryption. One can construct a single-server aggregation protocol using threshold additive homomorphic encryption [27,29,54,58,59,69]. Each client encrypts its vector as a ciphertext under the public key of the threshold scheme, and sends it to the committee. The committee adds the ciphertexts from all the clients and gives the result to the server. However, this does not work well for large inputs (like the large weight vectors found in federated learning) because encrypting the vector (say,

using Paillier or a lattice-based scheme) and performing the threshold decryption will be very expensive.

A recent work [67] uses LWE-based homomorphic PRGs. This is an elegant approach but it has higher computation and communication costs than works based on pairwise masking, including Flamingo. The higher cost stems from one having to choose parameters (e.g., vector length and the size of each component) that satisfy the LWE assumption, and particularly the large LWE modulus that is required.

Multi-round setting. Recent work [36] designs a new multi-round secure aggregation protocol with reusable secrets that is very different from Flamingo's design. The protocol works well for small input domains (e.g., vectors with small values) but cannot efficiently handle large domains as it requires brute forcing a discrete log during decryption. In contrast, Flamingo does not have any restriction on the input domain. A variant of the above work can also accommodate arbitrary input domains by relying on DDH-based class groups [19] (a more involved assumption than DDH).

11. Discussion

We have focused our discussion on computing sums, but Flamingo can also compute other functions such as max/min using affine aggregatable encodings [4,8,24,38].

Limitations. Flamingo assumes that the set of all clients (N) involved in a training session is fixed before the training starts and that in each round t some subset S_t from N is chosen. We have not yet explored the case of handling clients that dynamically join the training session.

Another aspect that we have not investigated in this work is that of handling an *adaptive* adversary that can dynamically change the set of parties that it compromises as the protocol executes. In BBGLR, the adversary can be adaptive across rounds but not within a round; in Flamingo the adversary is static across all the rounds. To our knowledge, an adversary that can be adaptive within a single round has not been considered before in the federating learning setting. It is not clear that existing approaches from other fields [34] can be used due to different communication models.

Finally, secure aggregation reduces the leakage of individuals' inputs in federated learning but does not fully eliminate it. It is important to understand what information continues to leak. Two recent works in this direction are as follows. Elkordy et al. [30] utilize tools from information theory to bound the leakage with secure aggregation: they found that the amount of leakage reduces linearly with the number of clients. Wang et al. [73] then show a new inference attack against federated learning systems that use secure aggregation in which they are able to obtain the proportion of different labels in the overall training data. While the scope of this attack is very limited, it may inspire more advanced attacks.

Acknowledgments

We thank the S&P reviewers for their comments which improved the content of this paper. We also thank Fabrice Benhamouda for discussion on elliptic curve scalar multiplication efficiency, Varun Chandrasekaran for advice on machine learning datasets, Yue Guo for answering questions about the ABIDES simulator, and Riad Wahby for pointers on hashing to elliptic curves. We thank Mariana Raykova and Adrià Gascón for helpful discussions about the BBGLR paper. Finally, we thank Dario Pasquini for making us aware of model inconsistency attacks in federated learning. This work was partially funded by NSF grant CNS-2045861, DARPA contract HR0011-17-C0047, and a JP Morgan Chase & Co Faculty Award.

References

[1] Cloudflare randomness beacon. https://developers.cloudflare.com/randomness-beacon/.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. Technical report, 2015. https://www.tensorflow.org/.

[3] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, and G. Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2023.

[4] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In C. Galdi and S. Jarecki, editors, *Proceedings of the International Conference on Security and Cryptography for Networks(SCN)*, 2022.

[5] E. Anderson, M. Chase, W. Dai, F. B. Durak, K. Laine, S. Sharma, and C. Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Paper 2021/1490, 2021. https://eprint.iacr.org/2021/1490.

[6] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for SNARKs. In *Proceedings of the USENIX Security Symposium*, 2022.

[7] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. In *Proceedings of the Artificial Intelligence and Statistics Conference (AISTATS)*, 2020.

[8] A. Beimel, A. Gabizon, Y. Ishai, E. Kushilevitz, S. Meldgaard, and A. Paskin-Cherniavsky. Non-interactive secure multiparty computation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2014.

[9] J. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[10] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun. Acorn: Input validation for secure aggregation. Cryptology ePrint Archive, Paper 2022/1461, 2022. https://eprint.iacr.org/2022/1461.

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

[12] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[13] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2010.

[14] D. Byrd, M. Hybinette, and T. H. Balch. ABIDES: Agent-based interactive discrete event simulation environment. https://github.com/abides-sim/abides, 2020.

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

[16] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečnỳ, H. B. McMahan, V. Smith, and A. Talwalkar. Leaf: A benchmark for federated settings. https://github.com/TalwalkarLab/leaf.

[17] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečnỳ, H. B. McMahan, V. Smith, and A. Talwalkar. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018. https://arxiv.org/abs/1812.01097.

[18] J. Canny and S. Sorkin. Practical large-scale distributed key generation. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2004.

[19] G. Castagnos and F. Laguillaumie. Linearly homomorphic encryption from ddh. Cryptology ePrint Archive, Paper 2015/047, 2015. https://eprint.iacr.org/2015/047.

[20] T.-H. H. Chan, E. Shi, and D. Song. Privacy-preserving stream aggregation with fault tolerance. In *Proceedings of the International Financial Cryptography Conference*, 2011.

[21] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[22] D. L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1), 1988.

[23] A. R. Chowdhury, C. Guo, S. Jha, and L. van der Maaten. Eiffel: Ensuring integrity for federated learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[24] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[25] S. Das, V. Krishnan, I. M. Isaac, and L. Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[26] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren. Practical asynchronous distributed key generation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[27] V. A. Dasu, S. Sarkar, and K. Mandal. PROV-FL: Privacy-preserving round optimal verifiable federated learning. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, 2022.

[28] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1989.

[29] T. Elahi, G. Danezis, and I. Goldberg. PrivEx: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[30] A. R. Elkordy, J. Zhang, Y. H. Ezzeldin, K. Psounis, and S. Avestimehr. How Much Privacy Does Federated Learning with Secure Aggregation Guarantee? In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2023.

[31] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1987.

[32] R. Gennaro, S. Halevi, H. Krawczyk, and T. Rabin. Threshold rsa for dynamic and adhoc groups. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2008.

[33] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Journal of Cryptology*, 2006.

[34] C. Gentry, S. Halevi, H. Krawczyk, B. Magri, J. B. Nielsen, T. Rabin, and S. Yakoubov. YOSO: You only speak once / secure MPC with stateless ephemeral roles. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2021.

[35] E. N. Gilbert. Random graphs. In *The Annals of Mathematical Statistics*, 1959.

[36] Y. Guo, A. Polychroniadou, E. Shi, D. Byrd, and T. Balch. MicroFedML: Privacy preserving federated learning for small weights. Cryptology ePrint Archive, Paper 2022/714, 2022. https://eprint.iacr.org/2022/714.

[37] S. Ha, I. Rhee, and L. Xu. Cubic: A new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 2008.

[38] S. Halevi, Y. Ishai, E. Kushilevitz, and T. Rabin. Best possible information-theoretic MPC. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2018.

[39] A. F. Hernández, S. Scott, N. Sullivan, R. S. Wahby, and C. Wood. Hashing to elliptic curves. https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-10.html, 2021.

[40] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or how to cope with perpetual leakage. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1995.

[41] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa. Merkle^2: A low-latency transparency log system. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[42] S. Kadhe, N. Rajaraman, O. O. Koyluoglu, and K. Ramchandran. FastSecAgg: Scalable secure aggregation for privacy-preserving federated learning. In *ICML Workshop on Federated Learning for User Privacy and Data Confidentiality*, 2020.

[43] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawit, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. El Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konecný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, H. Qi, D. Ramage, R. Raskar, M. Raykova, D. Song, W. Song, S. U. Stich, Z. Sun, A. Theertha Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. In *Foundations and Trends in Machine Learning*, 2021.

[44] E. Kokoris-Kogias, D. Malkhi, and A. Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[45] S. Krishna, D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song. Churp: Dynamic-committee proactive secret sharing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[46] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.
[47] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-100 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.
[48] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury. Oort: Efficient federated learning via guided participant selection. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
[49] D. Leung, Y. Gilad, S. Gorbunov, L. Reyzin, and N. Zeldovich. Aardvark: An asynchronous authenticated dictionary with short proofs. In *Proceedings of the USENIX Security Symposium*, 2022.
[50] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi. RoFL: Robustness of secure federated learning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023.
[51] M. Mansouri, M. Önen, W. B. Jaballah, and M. Conti. SoK: Secure aggregation based on cryptographic schemes for federated learning. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2023.
[52] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the Artificial Intelligence and Statistics Conference (AISTATS)*, 2017.
[53] S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In *Proceedings of the USENIX Security Symposium*, 2015.
[54] L. Melis, G. Danezis, and E. D. Cristofaro. Efficient private statistics with succinct sketches. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.
[55] L. Melis, C. Song, E. D. Cristofaro, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.
[56] D. Pasquini, D. Francati, and G. Ateniese. Eluding secure aggregation in federated learning via model inconsistency. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
[57] T. Pedersen. A threshold cryptosystem without a trusted party. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1991.
[58] R. A. Popa, H. Balakrishnan, and A. J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *Proceedings of the USENIX Security Symposium*, 2009.
[59] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.
[60] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
[61] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
[62] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
[63] E. Shi, T.-H. H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.
[64] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In *Journal of Cryptology*, 2002.
[65] J. So, B. Güler, and A. S. Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. In *Journal on Selected Areas in Information Theory*, 2021.
[66] J. So, C. He, C.-S. Yang, S. Li, Q. Yu, R. E. Ali, B. Guler, and S. Avestimehr. LightSecAgg: a lightweight and versatile design for secure aggregation in federated learning. In *Proceedings of Machine Learning and Systems*, 2022.
[67] T. Stevens, C. Skalka, C. Vincent, J. Ring, S. Clark, and J. Near. Efficient differentially private secure aggregation for federated learning via hardness of learning with errors. In *Proceedings of the USENIX Security Symposium*, 2022.
[68] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.
[69] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the ACM workshop on artificial intelligence and security*, 2019.
[70] N. Tyagi, B. Fisch, A. Zitek, J. Bonneau, and S. Tessaro. VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
[71] I. Tzialla, A. Kothapalli, B. Parno, and S. Setty. Transparency dictionaries with succinct proofs of correct operation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.
[72] R. S. Wahby and D. Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2019.
[73] L. Wang, S. Xu, X. Wang, and Q. Zhu. Eavesdrop the composition proportion of training labels in federated learning. arXiv:1910/06044, 2023. https://arxiv.org/abs/1910.06044.
[74] H. Yuan and T. Ma. Federated accelerated stochastic gradient descent. In *Neural Information Processing Systems (NeurIPS)*, 2020.
[75] K. Zhong, Y. Ma, and S. Angel. Ibex: Privacy-preserving ad conversion tracking and bidding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.
[76] K. Zhong, Y. Ma, Y. Mao, and S. Angel. Addax: A fast, private, and accountable ad exchange infrastructure. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
[77] L. Zhu, Z. Liu, and S. Han. Deep leakage from gradients. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Appendix A. Failure and threat model details

In this section, we give the full details of the dropout rate and the corruption rate (§2.3).

Dropout rate. Recall that we upper bound the dropout rate of the sum contributors (S_t) in one round as δ. For decryptors, we consider the dropout rate in one summation round and assume it is at most δ_D. Note that δ and δ_D are individually determined by the server timeout at those steps (recall that in each round, clients in S_t only participate in the first step; the following two steps only involve decryptors).

Corruption rate. For corruption, we denote the corrupted rate in S_t as η_{S_t} and the corrupted rate in decryptors as η_D. In the Flamingo system, η is given; η_{S_t} and η_D depends on η. Note that the fraction of malicious clients in a chosen subset of $[N]$ (e.g., S_t, $\mathcal{D}$) may not be exactly η, but rather a random variable η^* from a distribution that is parameterized by η, N and the size of the chosen set. Since the expectation of η^* is equal to η, and when the size of the chosen set is large (e.g., S_t), the deviation of η^* from η is negligible (i.e., η^* is almost equal to η). Therefore, η_{S_t} can be considered as equal to η. On the other hand, since $\mathcal{D}$ is a small set, we cannot assume η_D is equal to η. Later in Appendix C we

Protocol Π_{setup}.

Parties. Clients $1, \ldots, N$ and a server.

Parameters. Number of pre-selected decryptors L. Let $L = 3\ell + 1$.

Protocol outputs. A set of t clients ($2\ell + 1 \leq t \leq 3\ell + 1$) hold secret sharing of a secret key SK. All the clients in $[N]$ and the server hold the associated public key PK.

- The server and all the clients in $[N]$ invoke $\mathcal{F}_{\text{rand}}$ and receive a binary string $v \xleftarrow{\$} \{0,1\}^\lambda$.
- The server and all the clients in $[N]$ computes $\mathcal{D}_0 \leftarrow \text{CHOOSESET}(v, 0, L, N)$.
- All the clients $u \in \mathcal{D}_0$ and the server run Π_{DKG} (Fig. 12).
- The server broadcasts the signed PKs received from the clients in $\mathcal{D}_0$ to all the clients in $[N]$.
- A client in $[N]$ aborts if it received less than $2\ell+1$ valid signatures on PKs signed by the parties defined by $\text{CHOOSESET}(v, 0, L, N)$.

Figure 11: Setup phase with total number of clients N. $\mathcal{F}_{\text{rand}}$ is modeled as a random beacon service.

show how to choose L to ensure η_D satisfies the inequality required in Theorem 3 with overwhelming probability.

Security parameters. In the following definitions and proofs, we use κ for the information-theoretic security parameter and λ for the computational security parameter.

Appendix B. Full Protocol Description

B.1. Definition of cryptographic primitives

In this section, we formally define the cryptographic primitives used in Flamingo protocol that are not given in Section 3.1.

Definition 1 (DDH assumption)**.** Given a cyclic group $\mathbb{G}$ with order q, and let the generator of $\mathbb{G}$ be g. Let a, b, c be uniformly sampled elements from $\mathbb{Z}_q$. We say that DDH is hard if the two distributions (g^a, g^b, g^{ab}) and (g^a, g^b, g^c) are computationally indistinguishable.

Definition 2 (ElGamal encryption)**.** Let $\mathbb{G}$ be a group of order q in which DDH is hard. ElGamal encryption scheme consists of the following three algorithms.

- $\text{AsymGen}(1^\lambda) \to (SK, PK)$: sample a random element s from $\mathbb{Z}_q$, and output $SK = s$ and $PK = g^s$.
- $\text{AsymEnc}(PK, h) \to (c_0, c_1)$: sample a random element y from $\mathbb{Z}_q$ and compute $c_0 = g^y$ and $c_1 = h \cdot PK^y$.
- $\text{AsymDec}(SK, (c_0, c_1)) \to h$: compute $h = (c_0^{SK})^{-1} \cdot c_1$.

We say that ElGamal encryption is secure if it has CPA security. Note that if DDH assumption (Def.1) holds, then ElGamal encryption is secure.

Definition 3 (Authenticated encryption)**.** An authenticated encryption scheme consists of the following algorithms:

- $\text{SymAuthGen}(1^\lambda) \to k$: sample a key k uniformly random from $\{0,1\}^\lambda$.
- $\text{SymAuthEnc}(k, m) \to c$: take in a key k and a message m, output a ciphertext c.
- $\text{SymAuthDec}(k, c)$: take in a key k and a ciphertext c, output a plaintext m or $\perp$ (decryption fails).

We say that the scheme is secure if it has CPA security and ciphertext integrity.

For simplicity, we use AsymEnc and SymAuthEnc to refer to the encryption schemes.

Definition 4 (Signature scheme)**.** A signature scheme consists of the following algorithms:

- $SGen(1^\lambda) \to (sk, pk)$: generate a pair of siging key sk and verfication key pk.
- $Sign(sk, m) \to \sigma$: take in a signing key sk and message m, outputs a signature σ.
- $VerSig(pk, m, \sigma) \to b$: take in a verification key pk, a messagee m and a signature σ, output valid or not as $b = 1, 0$.

We say that the signature scheme is secure if the probability that, given $m_1, \ldots, m_z$, an attacker who can query the signing challenger and finds a valid (m', σ') where $m' \notin \{m_1, \ldots, m_z\}$ is negligible.

B.2. Setup phase and distributed key generation

The setup protocol is conceptually simple, as shown in Figure 11. A crucial part of the setup phase is the distributed key generation (DKG). We first describe the algorithms used in DKG.

Algorithms. Let $\mathbb{G}$ be a group with order q in which discrete log is hard. The discrete-log based DKG protocol builds on Feldman verifiable secret sharing [31], which we provide below. The sharing algorithm takes in the threshold parameters L, ℓ, and a secret $s \in \mathbb{Z}_q$, chooses a polynomial with random coefficients except the constant term, i.e., $p(X) = a_0 + a_1 X + \ldots + a_\ell X^\ell (a_0 = s)$, and outputs the commitments $A_k = g^{a_k} \in \mathbb{G}$ for $k = 0, 1, \ldots, \ell$. The j-th share s_j is $p(j)$ for $j = 1, \ldots, L$.

To verify the j-th share against the commitments, the verification algorithm takes in s_j and a set of commitments $\{A_k\}_{k=0}^\ell$, and checks if

$$g^{s_j} = \prod_{k=0}^{\ell} (A_k)^{j^k}.$$

We define the above algorithms as

- $FShare(s, \ell, L) \to \{s_j\}_{j=1}^L, \{A_k\}_{k=0}^\ell$,
- $FVerify(s_j, \{A_k\}_{k=0}^\ell) \to b$ where $b \in \{0, 1\}$.

The GJKR-DKG uses a variant of the above algorithm, $PShare$ and $PVerify$ based on Pedersen commitment, for security reason [33]. The $PShare$ algorithm chooses two random polynomials

$$p(X) = a_0 + a_1 X + \ldots + a_\ell X^\ell, \quad a_0 = s$$

$$p'(X) = b_0 + b_1 X + \ldots + b_\ell X^\ell$$

and outputs

$$\{p(j)\}_{j=1}^{L}, \{p'(j)\}_{j=1}^{L}, C_k := g^{a_k} h^{b_k} \text{ for } k = 0, \ldots, \ell,$$

where $g, h \in \mathbb{G}$.

To verify against the share $s_j = p(j)$, *PVerify* takes in $s'_j = p'(j)$ and $\{C_k\}_{k=0}^{\ell}$, and checks if

$$g^{s_j} h^{s'_j} = \prod_{k=0}^{\ell} (C_k)^{j^k}.$$

The algorithms *PShare* and *PVerify* can be defined analogously to *Fshare* and *FVerify*:

- $PShare(s, \ell, L) \to \{s_j\}_{j=1}^{L}, \{s'_j\}_{j=1}^{L}, \{C_k\}_{k=0}^{\ell}$,
- $PVerify(s_j, s'_j, \{C_k\}_{k=0}^{\ell}) \to b$ where $b \in \{0, 1\}$.

Protocol. We give the modified DKG protocol Π_{DKG} from GJKR-DKG in Figure 12. The participating parties can drop out, as long as $\eta_D + \delta_D < 1/3$.

Correctness and security. We analyze the properties of Π_{DKG} in this section. We start by revisiting the correctness and security definitions of GJKR-DKG, and then discuss how our definition differs from theirs because of a weakening of the communication model. In GJKR-DKG, correctness has three folds:

1) All subsets of honest parties define the same unique secret key.
2) All honest parties have the same public key.
3) The secret key is uniformly random.

Security means that no information about the secret key can be learned by the adversary except for what is implied by the public key. For Π_{DKG}, if the server is honest, then our communication model (§2.3) is equivalent to having a fully synchronous channel, hence in this case the correctness and security properties in the prior work hold. When the server is malicious, we show that Π_{DKG} satisfies the following correctness (C1, C2, C3, C4) and security (S).

- C1. Each honest party either has no secret at the end or agrees on the same QUAL with other honest parties.
- C2. The agreed QUAL sets defines a unique secret key.
- C3. The secret key defined by QUAL is uniformly random.
- C4. Each honest party, either has no public key, or outputs the same public key with other honest parties.
- S. Malicious parties learns no information about the secret key except for what is implied by the public key.

Lemma 2. Let the participants in DKG be L parties and a server. If $\delta_D + \eta_D < 1/3$, then under the communication model defined in Section 2.2, protocol Π_{DKG} (Fig. 12) has properties C1, C2, C3, C4 and S in the presence of a malicious adversary controlling the server and up to η_D fraction of the parties.

Proof. Since $L = 3\ell+1$, and by $\delta_D+\eta_D < 1/3$, at most ℓ are malicious (the dropouts can be treated as malicious parties who do not send the prescribed messages). We first show under which cases the honest parties will have no share. Note that the parties that are excluded from $\mathcal{D}_2$ are those who are honest but did not receive a complaint against a malicious party who performed wrong sharing; and parties that are excluded from $\mathcal{D}_3$ are those who have complained at i in step (b) but did not receive shares from i at this step. If the server drops messages (sent from one online honest party to another) in the above two cases, then in the cross-check step, the honest parties will not receive more than $2\ell + 1$ QUALs, and hence will abort. In this case, honest parties in the end has no share.

Now we prove C1 by contradiction. Suppose there are two honest parties P_1 and P_2 at the end of the protocol who holds secrets (not abort) and they have different QUAL. Then this means P_1 received at least $2\ell + 1$ same QUAL sets S_1 and P_2 received at least same $2\ell + 1$ QUAL sets S_2. W.l.o.g., assume that there are $\ell - v$ malicious parties ($v \geq 0$). In the $2\ell + 1$ sets S_1, at least $\ell + 1 + v$ of them are from honest parties. Similarly, for the $2\ell + 1$ sets S_2, at least $\ell + 1 + v$ are from other honest parties different than above (since an honest party cannot send both S_1 and S_2). However, note that we have in total $2\ell+1+v$ honest parties, which derives a contradiction.

Recall that at most ℓ parties are malicious, so the QUAL set has at least $\ell + 1$ parties, and since we have C1, now C2 is guaranteed. Moreover, since QUAL contains at least one honest party, the secret key is uniform (C3). C4 follows exactly from the work GJKR. The proof for S is the same as GKJR, except that the simulator additionally simulates the agreement protocol in Lemma 3.

Remark. An important difference between Π_{DKG} and standard DKG protocols is that we allow aborts and allow honest parties to not have shares at the end of the protocol. When some honest parties do not have shares of the secret key, the server is still able to get sum (decryption still works) because malicious parties hold the shares.

B.3. Collection phase

The detailed protocol for each round in the collection phase is described in Figure 13. At the beginning of round t, the server notifies the clients who should be involved, namely S_t. A client who gets a notification can download public keys of its neighbors $A_t(i)$ from PKI server (the server should tell clients how to map client IDs to the names in PKI). To reduce the overall run time, clients can pre-fetch public keys used in the coming rounds.

B.4. Transfer shares

Every R rounds, the current set of decryptors $\mathcal{D}$ transfer shares of SK to a new set of decryptors, $\mathcal{D}_{new}$. To do so, each $u \in \mathcal{D}$ computes a destination decryptors set $\mathcal{D}_{new}$ for round t, by $\text{CHOOSESET}(v, \lceil t/R \rceil, L, N)$. Assume now each decryptor $u \in \mathcal{D}$ holds share s_u of SK (i.e., there is

Protocol Π_{DKG} based on discrete log

Parameters. A set of L parties (denoted as $\mathcal{D}_0$), threshold ℓ where $3\ell + 1 = L$. $\delta_D + \eta_D < 1/3$.

Protocol outputs. A subset of the L parties hold secret sharing of a secret key $s \in \mathbb{Z}_q$; the server holds the public key g^s signed by all the clients.

Notes. The parties have access to PKI (Section 4.2). All messages sent from one party to another via the server are signed and end-to-end encrypted.

1. Each party i performs verifiable secret sharing (VSS) as a dealer:
   a) *Share*:
      Party $i \in \mathcal{D}_0$ randomly chooses $s_i \in \mathbb{Z}_q$, computes $\{s_{i,j}\}_{j=1}^{L}, \{s'_{i,j}\}_{j=1}^{L}, \{C_{i,k}\}_{k=0}^{\ell} \leftarrow \mathit{PShare}(s_i, \ell, L)$.
      It also computes $\{A_k\}_{k=0}^{\ell}$ from $\mathit{FShare}(s, \ell, L)$ and stores it locally.
      Send $s_{i,j}$ and $s'_{i,j}$ to each party j, and $\{C_{i,k}\}_{i=0}^{\ell}$ to all parties $j \in \mathcal{D}_0$ via the server.
      // Denote the set of parties who received all the prescribed messages after this step as $\mathcal{D}_1$.
   b) *Verify and complain*:
      Each party $j \in \mathcal{D}_1$ checks whether it received at least $(1 - \delta_D)L$ valid signed shares. If not, abort; otherwise continue.
      Each party $j \in \mathcal{D}_1$, for each received share $s_{i,j}$, runs $b \leftarrow \mathit{PVerify}(j, s_{i,j}, s'_{i,j}, \{C_{i,k}\}_{k=0}^{\ell})$.
      If b is 1, then party i does nothing; otherwise party i sends (`complaints`, j) to all the parties in $\mathcal{D}_0$ via the server.
      // Denote the set of parties who received all the prescribed messages after this step as $\mathcal{D}_2$.
   c) *Against complaint*:
      Each party $i \in \mathcal{D}_2$, who as a dealer, if received a valid signed (`complaint`, i) from j, sends $s_{i,j}, s'_{i,j}$ to all parties in $\mathcal{D}_0$ via the server.
      // Denote the set of parties who received all the prescribed messages after this step as $\mathcal{D}_3$.
   d) *Disqualify*:
      Each party $i \in \mathcal{D}_3$ marks any party j as disqualified if it received more than $2\ell + 1$ valid signed (`complaints`, j), or party j answers with $s_{j,i}, s'_{j,i}$ such that $\mathit{PVerify}(s_{j,i}, s'_{j,i}, \{C_{j,k}\}_{k=1}^{\ell})$ outputs 0. The non-disqualified parties form a set QUAL.
      Each party $i \in \mathcal{D}_3$ signs the QUAL set and sends to all parties in $\mathcal{D}_0$ to the server. The server, on receiving a valid signed QUAL, signs and sends it to all parties in $\mathcal{D}_3$.
   e) *Cross-check QUAL*:
      Each party $i \in \mathcal{D}_3$ checks whether it receives at least $2\ell + 1$ valid signed QUAL, if so, they sum up the shares in QUAL and derive a share of secret key. If not, abort.
2. Compute public key: // Each party exposes g^{s_i} as shares of PK
   a) Each party $i \in$ QUAL sends $\{A_{i,k}\}_{k=1}^{\ell}$ to all parties via the server.
   b) Each party i runs $b' \leftarrow \mathit{FVerify}(s_{j,i}, \{A_{j,k}\}_{k=1}^{\ell})$ for $j \in$ QUAL. If b' is 0, then party i sends to all the parties in $\mathcal{D}_3 \cap$ QUAL via the server a message (`complaint`, $j, s_{j,i}, s'_{j,i}$) for those $(s_{j,i}, s'_{j,i})$ such that b is 1 and b' is 0.
      // For $b = 1$ and $b' = 0$: The check in step 1.d) passes but fails this step
   c) For each j such that (`complaint`, $j, s_{j,i}, s'_{j,i}$) is valid, parties reconstruct s_j. For all parties in QUAL, set $y_i = g^{s_i}$, and compute $PK = \prod_{i \in QUAL} y_i$. Parties in QUAL sign PK using their own signing keys and send the signed PK to the server.

Figure 12: Protocol Π_{DKG}.

a polynomial p such that $p(u) = s_u$ and $p(0) = SK$). To transfer its share, each $u \in \mathcal{D}$ acts as a VSS dealer exactly the same as the first part in Π_{DKG} (Fig.9) to share s_u to new decryptor $j \in \mathcal{D}_{new}$. In detail, u chooses a polynomial p^*_u of degree ℓ and sets $p^*_u(0) = s_u$ and all other coefficients of p^*_u to be random. Then, u sends $p^*_u(j)$ to each new decryptor $j \in \mathcal{D}_{new}$.

Each new decryptor $j \in \mathcal{D}_{new}$ receives the evaluation of the polynomials at point j (i.e., $p^*_u(j)$ for all $u \in \mathcal{D}$). The new share of SK held by j, s'_j, is defined to be a linear combination of the received shares: $s'_j := \sum_{u \in \mathcal{D}} \beta_u \cdot p^*_u(j)$, where the combination coefficients $\{\beta_u\}_{u \in \mathcal{D}}$ are constants (given the set $\mathcal{D}$, we can compute $\{\beta_u\}_{u \in \mathcal{D}}$). Note that the same issue about communication model for DKG also exists here, but the same relaxation applies.

Here we require that $\eta_D + \delta_D < 1/3$ for both $\mathcal{D}$ and $\mathcal{D}_{new}$. As a result, each client j in a subset $\mathcal{D} \subseteq \mathcal{D}_{new}$ holds a share $s_{u,j}$. For each receiving decryptor $j \in \mathcal{D}$, it computes $s'_j = \sum_{u \in \mathcal{D}_{old}} \beta_u \cdot s_{u,j}$, where each β_u is some fixed constant.

Since the sharing part is exactly the same as Π_{DKG} and the share combination happens locally, the same correctness and security argument of DKG applies. Specifically, for correctness, each honest party either has no secret at the end or agrees on the same QUAL with other honest parties; and the QUAL defines the unique same secret key before resharing. For security, a malicious adversary will not learn any information, but can cause the protocol aborts.

Collection phase: Π_{sum} for round t out of T total rounds

Initial state from setup phase: each client $i \in [N]$ holds a value v and public key $PK = g^s$ where $SK = s$; each decryptor $u \in \mathcal{D}$ additionally holds a Shamir share of SK (threshold ℓ with $3\ell + 1 = L$). We require $\delta_D + \eta_D < 1/3$.

Steps for round t:

1. **Report step.**

   Server performs the following:

   Compute a set $\mathcal{Q}_{graph} \leftarrow \text{CHOOSESET}(v, t, n_t, N)$ and a graph $G_t \leftarrow \text{GENGRAPH}(v, t, \mathcal{Q}_{graph})$; store $\{A_t(i)\}_{i \in \mathcal{Q}_{graph}}$ computed from G_t.

   Notify each client $i \in \mathcal{Q}_{graph}$ that collection round t begins.

   Each client $i \in \mathcal{Q}_{graph}$ performs the following:

   Compute $\mathcal{Q}^{local}_{graph} \leftarrow \text{CHOOSESET}(v, t, n_t, N)$, and if $i \notin \mathcal{Q}^{local}_{graph}$, ignore this round.

   Read from PKI g^{a_j} for $j \in A_t(i)$, and compute $r_{i,j}$ by computing $(g^{a_j})^{a_i}$ and mapping it to $\{0,1\}^\lambda$.

   Sample $m_{i,t} \xleftarrow{\$} \{0,1\}^\lambda$ and compute $\{h_{i,j,t}\}_{j \in A(i)} \leftarrow \text{PRF}(r_{ij}, t)$ for $j \in A_t(i)$.

   Send to server a message $msg_{i,t}$ consisting of

   $$Vec_{i,t} = \vec{x}_{i,t} + \text{PRG}(m_{i,t}) + \textstyle\sum_{j \in A_t(i)} \pm\text{PRG}(h_{i,j,t}), \quad \text{SymAuthEnc}(k_{i,u}, m_{i,u,t} \| t), \text{ for } u \in \mathcal{D}, \quad \text{AsymEnc}(PK, h_{i,j,t}) \text{ for } j \in A_t(i)$$

   where $m_{i,u,t} \leftarrow Share(m_{i,t}, \ell, L)$, $A_t(i) \leftarrow \text{FINDNEIGHBORS}(v, S_t, i)$,

   and AsymEnc (ElGamal) and SymAuthEnc (authenticated encryption) are defined in Appendix B.1.

   along with the signatures $\sigma_{i,j,t} \leftarrow Sign(sk_i, c_{i,j,t} \| t)$ for all ciphertext $c_{i,j,t} = \text{AsymEnc}(PK, h_{i,j,t})$ $\forall j \in A_t(i)$.

2. **Cross check step.**

   Server performs the following:

   Denote the set of clients that respond within timeout as $\mathcal{Q}_{vec}$.

   Compute partial sum $\tilde{z}_t = \sum_{i \in \mathcal{Q}_{vec}} Vec_{i,t}$.

   Build decryption request req (req consists of clients in S_t to be labeled):

   Initialize an empty set $\mathcal{E}_i$ for each $i \in \mathcal{Q}_{graph}$, and

   if $i \in \mathcal{Q}_{vec}$, label i with "online",

   and add $\text{SymAuthEnc}(k_{i,u}, m_{i,u,t} \| t)$ to $\mathcal{E}_i$, where $k_{i,j}$ is derived from PKI (Appendix B.1);

   else label i with "offline",

   and add $\{(\text{AsymEnc}(PK, h_{i,j,t}), \sigma_{i,j,t}\}_{j \in A_t(i) \cap \mathcal{Q}_{vec})}$ to $\mathcal{E}_i$.

   Send to each $u \in \mathcal{D}$ the request req and $\mathcal{E}_i$ of all clients $i \in \mathcal{Q}_{graph}$.

   Each decryptor $u \in \mathcal{D}$ performs the following:

   Upon receiving a request req, compute $\sigma^*_u \leftarrow Sign(sk_u, req \| t)$, and send (req, σ^*_u) to all other decryptors via the server.

3. **Reconstruction step.**

   Each decryptor $u \in \mathcal{D}$ performs the following:

   Ignore messages with signatures ($\sigma_{i,j,t}$ or σ^*_u) with round number other than t.

   Upon receiving a message $(req, \sigma^*_{u'})$, run $b \leftarrow VerSig(pk_i, req, \sigma^*_{u'})$. Ignore the message if $b = 0$.

   Continue only if u received $2\ell + 1$ or more same req messages that were not ignored. Denote such message as req^*.

   For req^*, continue only if

   each client $i \in S_t$ is either labeled as "online" or "offline";

   the number of "online" clients is at least $(1 - \delta)n_t$;

   all the "online" clients are connected in the graph;

   each online client i has at least k online neighbors such that $\eta^k < 2^{-\kappa}$.

   For each $i \in \mathcal{Q}_{graph}$,

   For each $\text{SymAuthEnc}(k_{i,u}, m_{i,u,t} \| t)$ in $\mathcal{E}_i$, use $k_{i,u}$ (derived from PKI) to decrypt; send $m_{i,u,t}$ to the server if the decryption succeeds;

   For each $(\text{AsymEnc}(PK, h_{i,j,t}), \sigma_{i,j,t}) \in \mathcal{E}_i$, parse as $((c_0, c_1), \sigma)$ and send $c_0^{s_u}$ to the server if $VerSig((c_0, c_1), \sigma)$ outputs 1;

   Server completes the sum:

   Denote the set of decryptors whose messages have been received as U. Compute a set of interpolation coefficients $\{\beta_u\}_{u \in U}$ from U.

   For each $i \in \mathcal{Q}_{graph}$, reconstruct the mask $m_{i,t}$ or $\{h_{i,j,t}\}_{j \in A_t(i) \cap \mathcal{Q}_{vec}}$:

   For each parsed (c_0, c_1) meant for $h_{i,j,t}$ in $\mathcal{E}_i$, compute $h_{i,j,t}$ as $c_1 \cdot (\prod_{u \in U} (c_0^{s_u})^{\beta_u})^{-1}$;

   For each set of received shares $\{m_{i,u,t}\}_{u \in U}$, compute $m_{i,t}$ as $Recon(\{m_{i,u,t}\}_{u \in U})$.

   Output $z_t = \tilde{z}_t - \text{PRG}(m_{i,t}) + \sum_{j \in A_t(i) \cap \mathcal{Q}_{vec}} \pm\text{PRG}(h_{i,j,t})$.

Figure 13: Collection protocol Π_{sum}.

Functionality $\mathcal{F}_{\text{sum}}^{(t)}$

Parties: clients in S_t and a server.

Parameters: dropout rate δ and malicious rate η over n_t clients.

- $\mathcal{F}_{\text{sum}}^{(t)}$ receives a set $\mathcal{O}_t$ such that $|\mathcal{O}_t|/|S_t| \leq \delta$, and from the adversary $\mathcal{A}$ a set of corrupted parties, $\mathcal{C}$; and $\vec{x}_{i,t}$ from client $i \in S_t \backslash (\mathcal{O}_t \cup \mathcal{C})$.
- $\mathcal{F}_{\text{sum}}^{(t)}$ asks $\mathcal{A}$ for a set: if $\mathcal{A}$ replies with a set $M_t \subseteq S_t \backslash \mathcal{O}_t$ such that $|M_t|/|S_t| \geq 1-\delta$, then $\mathcal{F}_{\text{sum}}^{(t)}$ outputs $z'_t = \sum_{i \in M_t \backslash \mathcal{C}} \vec{x}_{i,t}$; otherwise sends `abort` to all honest clients.

Figure 15: Ideal functionality for round t in collection phase.

Appendix C. Requirements on Parameters

C.1. The number of decryptors

In this section, we show how to choose L such that, given N, η, δ_D, we can guarantee less than $1/3$ of the L chosen decryptors are malicious. Note that δ_D is given because this can be controlled by the server, i.e., the server can decide how long to wait for the decryptors to respond.

To guarantee $2\delta_D + \eta_D < 1/3$ (Theorem 3), a necessary condition is that $\eta < 1/3 - 2\delta_D$. This can be formalized as a probability question: Given N clients where η fraction of them are malicious, and randomly choose L clients (decryptors) from them; the malicious clients X in decryptors can be upper bounded using Chernoff bound as

$$\Pr[X \geq (\eta + (1/3 - 2\delta_D - \eta))L] \leq e^{-2L(1/3-\eta-2\delta_D)^2},$$

For η and δ_D both being 1%, the choice of $L = 60$ (Section 6) gives 10^{-6} probability. If we double L to be 120, then this guarantees 10^{-12} probability.

C.2. Proof of Lemma 1

The algorithm in Figure 1 gives a random graph $G(n, \epsilon)$. A known result in random graphs is, when the edge probability $\epsilon > \frac{(1+\omega)\ln n}{n}$, where ω is an arbitrary positive value, the graph is almost surely connected when n is large. Note that in BBGLR, they also use this observation to build the graph that has significant less number of neighbors than a prior work by Bonawitz et al. [11], but in their work since the clients chooses their neighbors, the resulting graph is a biased one; and they guarantee that there is no small isolated components in the graph.

Concretely, from Gilbert [35], let $g(n, \epsilon)$ be the probability that graph $G(n, \epsilon)$ has disconnected components, and it can be recursively computed as

$$g(n, \epsilon) = 1 - \sum_{i=1}^{n-1} \binom{n-1}{i-1} g(i, \epsilon)(1-\epsilon)^{i(n-i)}.$$

Number of nodes n	128	512	1024
Parameter ϵ (failure probability 10^{-6})	0.11	0.03	0.02
Parameter ϵ (failure probability 10^{-12})	0.25	0.06	0.03

Figure 14: Parameters ϵ to ensure random graph connectivity.

We numerically depict the above upper bound of the probability $g(n, \epsilon)$ for different ϵ in Figure 14. For example, when $n = 1024$, to ensure less than 10^{-6} failure probability, we need $\epsilon \geq 0.02$, hence the number of neighbors a client needs when $\delta = \eta = 0.01$ is at least $\lceil (\epsilon + \delta + \eta)n \rceil = 41$. Recall that when η and δ_D are both 1%, the lower bound for number of decryptors L is roughly 60 (Appendix C.1).

Appendix D. Security Proofs

D.1. Security definition

We say *a protocol* Π *securely realizes ideal functionality* $\mathcal{F}$ *in the presence of a malicious adversary* $\mathcal{A}$ if there exists a probabilistic polynomial time algorithm, or simulator, $\mathcal{S}$, such that for any probabilistic polynomial time $\mathcal{A}$, the distribution of the output of the real-world execution of Π is (statistically or computationally) indistinguishable from the distribution of the output of the ideal-world execution invoking $\mathcal{F}$: the output of both worlds' execution includes the inputs and outputs of honest parties the view of the adversary $\mathcal{A}$. In our proof, we focus on the computational notion of security. Note that $\mathcal{S}$ in the ideal world has one-time access to $\mathcal{F}$, and has the inputs of the corrupted parties controlled by $\mathcal{A}$.

D.2. Ideal functionalities

We provide ideal functionality for Flamingo in Figure 4. Looking ahead in the proof, we also need to define an ideal functionality for the setup phase and an ideal functionality for a single round in the collection phase, which we give as Figure 3 and Figure 15.

We model the trusted source of initial randomness as a functionality $\mathcal{F}_{\text{rand}}$; that is, a party or ideal functionality on calling $\mathcal{F}_{\text{rand}}$ will receive a uniform random value v in $\{0,1\}^\lambda$, where λ is the length of the PRG seed.

D.3. Proof of Theorem 1

The ideal functionality $\mathcal{F}_{\text{setup}}$ for the setup phase is defined in Figure 3. Depending on whether the server is corrupted or not, we have the following two cases.

1) When the server is not corrupted, then the communication model is equivalent to a secure broadcast channel. By security of GJKR, Π_{setup} securely realizes $\mathcal{F}_{\text{setup}}$.
2) When the server is corrupted, we build a simulator for the adversary $\mathcal{A}$. We start by listing the messages that $\mathcal{A}$ sees throughout the setup phase:

- A random value v from $\mathcal{F}_{\text{rand}}$;
- All the messages in Π_{DKG} that are sent via the server;
- All the messages in Π_{DKG} that are seen by the corrupted decryptors.

The simulator first calls $\mathcal{F}_{\text{rand}}$ and receives a value v. Then the simulator interacts with $\mathcal{A}$ acting as the honest decryptors. The simulator aborts if any honest decryptors would abort in Π_{DKG}. There are two ways that $\mathcal{A}$ can cheat: 1) $\mathcal{A}$ cheats in Π_{DKG}, and in Appendix B.2, we show the simulator can simulate the view of $\mathcal{A}$; 2) $\mathcal{A}$ cheats outside Π_{DKG}, this means $\mathcal{A}$ chooses a wrong set of decryptors, or it broadcasts wrong signatures. So the simulator aborts as long as it does not receive $2\ell+1$ valid signatures on PKs signed by the set defined by v.

Finally, note that our threat model (§2.3) assumes that the server is also controlled by the adversary, i.e., the first case is not needed here; but it will be useful when we analyze the robust version in Appendix E.1.

D.4. Proof of Theorem 2

The proof for dropout resilience is rather simple: in the setup phase, at most δ_D fraction of L selected decryptors drop out; then in one round of the collection phase, another δ_D fraction of decryptors can drop out. Since $2\delta_D + \eta_D < 1/3$, and $3\ell + 1 = L$ (Fig. 12), the online decryptors can always help the server to reconstruct the secrets.

D.5. Proof of Theorem 3

We first present the proof for a single round: the collection protocol Π_{sum} (Fig. 13) for round t securely realizes the ideal functionality $\mathcal{F}_{\text{sum}}^{(t)}$ (Fig. 15) in the random oracle model. From the ideal functionality $\mathcal{F}_{\text{sum}}^{(t)}$ we can see that when the server is corrupted, the result is not determined by the actual dropout set $\mathcal{O}_t$, but instead M_t sent by the adversary; when $M_t = S_t \backslash \mathcal{O}_t$ (in this case the adversary does not cheat or the cheating behavior does not affect the output), z'_t learned by the adversary in $\mathcal{F}_{\text{sum}}^{(t)}$ equals z_t computed in the protocol Π_{sum}. In the robust version (Appendix E.1), we will see that when the server is honest, the result is determined by $\mathcal{O}_t$.

In the proof below for a single round, for simplicity, we omit the round number t when there is no ambiguity. We assume the adversary $\mathcal{A}$ controls a set of clients in $[N]$, with the constraint $2\delta_D + \eta_D < 1/3$. Denote the set of corrupted clients in $[N]$ as $\mathcal{C}$ and as before the set of the decryptors is $\mathcal{D}$; the malicious decryptors form a set $\mathcal{C} \cap \mathcal{D}$ and $|\mathcal{C} \cap \mathcal{D}| < L/3$. From the analysis in Appendix A, we have $|\mathcal{C}|/|S_t| \leq \eta$.

Now we construct a simulator $\mathcal{S}$ in the ideal world that runs $\mathcal{A}$ as a subroutine. We assume both the simulator $\mathcal{S}$ and ideal functionality $\mathcal{F}_{\text{sum}}^{(t)}$ (Fig. 15) have access to an oracle $\mathcal{R}_{\text{drop}}$ that provides the dropout sets $\mathcal{O}_t$. In other words, the dropout set is not provided ahead of the protocol but instead provided during the execution (similar notion appeared in prior work [11]). Assume that in the ideal world, initially a secret key SK is shared among at least $2L/3$ clients in $\mathcal{D}$. The simulation for round t is as follows.

1) $\mathcal{S}$ received a set $\mathcal{O}_t$ from the oracle $\mathcal{R}_{\text{drop}}$.
2) $\mathcal{S}$ receives a set M_t from the adversary $\mathcal{A}$.
3) $\mathcal{S}$ obtains z'_t from $\mathcal{F}_{\text{sum}}^{(t)}$.
4) (Report step) $\mathcal{S}$ interacts with $\mathcal{A}$ as in the report step acting as the honest clients $i \in M_t \backslash \mathcal{C}$ with masked inputs $\vec{x}'_i$, such that $\sum_{i \in M_t \backslash \mathcal{C}} \vec{x}'_i = z'_t$.
   Here the input vector $\vec{x}'_i$ and the mask m_i are generated by $\mathcal{S}$ itself, and the pairwise secrets are obtained by querying the PKI.
5) (Cross-check step) $\mathcal{S}$ interacts with $\mathcal{A}$ acting as honest decryptors as in the cross-check step.
6) (Reconstruction step) $\mathcal{S}$ interacts with $\mathcal{A}$ acting as honest decryptors in the reconstruction step, where $\mathcal{S}$ uses the shares of the secret key SK to perform decryption of honest parties.
7) In the above steps, if all the honest decryptors would abort in the protocol prescription then $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{sum}}^{(t)}$, outputs whatever $\mathcal{A}$ outputs and halts.

We construct a series of hybrid execution, starting from the real world to the ideal world execution.

Hybrid 1. The view of $\mathcal{A}$ in the real world execution is the same as the view of $\mathcal{A}$ in the ideal world when $\mathcal{S}$ would have the actual inputs of honest parties, $\{\vec{x}_i\}_{i \in S_t \backslash (\mathcal{C} \cup \mathcal{O}_t)}$, the pairwise and individual masks, and the shares of the secret key SK. ($\mathcal{S}$ in fact would know SK in full because $3\ell+1 = L$ and that the number of honest parties is $2\ell + 1$ or more.)

Hybrid 2. $\mathcal{S}$ now instead of using the actual secret key s, it replaces s with 0 and sends the corresponding $|\mathcal{C} \cap \mathcal{D}| < L/3$ shares of 0 in $\mathbb{Z}_q$ to $\mathcal{A}$. The joint distribution of less than $L/3$ shares (recall that the threshold is ℓ where $3\ell + 1 = L$), from the property of Shamir secret sharing, for s and 0 are the same. Hence this hybrid has identical distribution to the previous hybrid.

Hybrid 3. $\mathcal{S}$ now instead of using the actual pairwise masks between honest parties, it samples a random pairwise mask $r'_{i,j}$ from $\{0,1\}^\lambda$ and computes the corresponding ElGamal ciphertext as (c'_0, c'_1). $\mathcal{S}$ does not change the pairwise mask between a client controlled by $\mathcal{A}$ and an honest client (such pairwise mask can be obtained by querying PKI to get g^{a_j} for an honest client j, and compute $(g^{a_j})^{a_i}$ for malicious client i). We argue that $\mathcal{A}$'s view in this hybrid is computationally indistinguishable from the previous one as below.

First, we need to assume the mapping from $\mathbb{G}$ to $\{0,1\}^\lambda$ is a random oracle. To specify, in the real world, the mask $r_{i,j}$ is computed from the mapping on $g^{a_i a_j}$; and in the ideal world the mask $r'_{i,j}$ is randomly sampled. Let M_t be the set of online clients that $\mathcal{A}$ labels in the real world (recall the server is controlled by $\mathcal{A}$). $\mathcal{A}$ in both worlds observes $\text{PRF}(r_{i,j}, t)$ between a client i out of M_t and a client j in M_t, hence we require $r_{i,j}$ to be random as a PRF key.

Second, $\mathcal{A}$ in the ideal world does not observe the pairwise masks between clients in M_t, but only the ciphertexts generated from $r'_{i,j}$ for those clients; and the distribution of the ciphertexts is computationally indistinguishable from

what $\mathcal{A}$ observed from the real world by the security of ElGamal encryption (Definition 2).

Hybrid 4. $\mathcal{S}$ now instead of using symmetric encryption (SymAuthEnc) of the shares of the actual individual mask m_i, it uses the symmetric encryption of a randomly sampled m_i' from $\{0,1\}^\lambda$ as the individual mask. Looking ahead in the proof, we also need to model the PRG as a random oracle $\mathcal{R}_{\text{PRG}}$ that can be thought of as a "perfect PRG" (see more details in a prior work [11]). For all $i \in M_t/\mathcal{C}$, $\mathcal{S}$ samples Vec_i at random and programs $\mathcal{R}_{\text{PRG}}$ to set $\text{PRG}(m_i')$ as

$$\text{PRG}(m_i') = Vec_i - \vec{x}_i - \sum \pm \text{PRG}(r_{i,j}'),$$

where the vectors Vec_i's are vectors observed in the real world. The view of $\mathcal{A}$ regarding Vec_i's in this hybrid is statistically indistinguishable to that in the previous hybrid.

Moreover, $\mathcal{A}$ learns the m_i in the clear for those $i \in M_t$ in both worlds, and the distributions of those m_i's in the ideal and real world are identical; for those m_i's where $i \notin M_t$ that $\mathcal{A}$ should not learn, from the semantic security of the symmetric encryption scheme (Definition 3), and the threshold $\ell < L/3$, $\mathcal{A}$'s view in this hybrid is computationally indistinguishable from the previous one.

Hybrid 5. $\mathcal{S}$ now instead of programming the oracle $\mathcal{R}_{\text{PRG}}$ as in the previous hybrid, it programs the oracle as

$$\text{PRG}(m_i') = Vec_i - \vec{x}_i' - \sum \pm \text{PRG}(r_{i,j}'),$$

where $\vec{x}_i'$'s are chosen such that $\sum_{i \in M_t \setminus \mathcal{C}} \vec{x}_i = \sum_{i \in M_t \setminus \mathcal{C}} \vec{x}_i'$. From Lemma 6.1 in a prior work [11] under the same setting, the distribution of $\mathcal{A}$'s view in this hybrid is statistically indistinguishable to that in the previous hybrid, except probability $2^{-\kappa}$: if the graph becomes disconnected or there is an isolated node after removing $\mathcal{O}_t$ and $\mathcal{C}$ from S_t, then the server learns x_i in the clear and thus can distinguish between the two worlds. When $\mathcal{A}$ cheats by submitting M_t to $\mathcal{S}$ where the graph formed by nodes in M_t is not connected, $\mathcal{S}$ simulates honest decryptors and output abort. In this case, the distribution of $\mathcal{A}'s$ view in the ideal world is the same as that in the real world.

Hybrid 7. Same as the previous hybrid, except that the label messages *req* from honest decryptors in the last step are replaced with the offline/online labels obtained from the oracle. In all steps, $\mathcal{A}$ would cheat by sending invalid signatures to $\mathcal{S}$; in this case $\mathcal{S}$ will abort. In the cross-check and reconstruction steps, there are following ways that $\mathcal{A}$ would cheat here:

1) $\mathcal{A}$ sends multiple different M_t's to $\mathcal{F}_{\text{sum}}$. $\mathcal{S}$ in the ideal world will simulate the protocol in Lemma 3 below, and outputs whatever the protocol outputs.
2) $\mathcal{A}$ sends to $\mathcal{F}_{\text{sum}}$ a set M_t with less than $(1-\delta)n_t$ clients, or the clients in M_t are disconnected, or there is a client in M_t with less than η^k online neighbors. In this case, $\mathcal{S}$ will abort, which is the same as in the real-world execution.

The last hybrid is exactly the ideal world execution. To better analyze the simulation succeeding probability, we use κ_1 to denote the security parameter for the graph connectivity (Lemma 1) and use κ_2 to denote the security parameter for the third checking in the cross-check round (§4.4). The simulation can fail in two ways: 1) The graph gets disconnected (even when the server is honest); 2) There exists a client in S_t such that all of its online neighbors are malicious. The former happens with probability $2^{-\kappa_1}$. The latter is bounded by $n \cdot 2^{-\kappa_2}$: the probability that the opposite event of 2) happens is $(1-\eta^k)^n \approx 1 - n\eta^k$ (assuming η^k is very small). Thus the failure probability $n\eta^k \le n \cdot 2^{-\kappa_2}$. This completes the proof that for any single round $t \in [T]$, the protocol Π_{sum} for round t securely realizes $\mathcal{F}_{\text{sum}}^{(t)}$ when $\delta_D + \eta_D < 1/3$, except probability $2^{-\kappa_1} + n \cdot 2^{-\kappa_2} \le n \cdot 2^{-\kappa+1}$, where $\kappa = \min\{\kappa_1, \kappa_2\}$.

Lemma 3. Assume there exists a PKI and a secure signature scheme; there are $3\ell+1$ parties with at most ℓ colluding malicious parties. Each party has an input bit of 0 or 1 from a server. Then there exists a one-round protocol for honest parties to decide if the server sent the same value to all the honest parties.

Proof. If an honest party receives $2\ell+1$ or more messages with the same value, then it means the server sends to all honest parties the same value. If an honest party receives less than $2\ell+1$ messages with the same value, it will abort; in this case the server must have sent different messages to different honest parties. □

Remark. The above analysis of the agreement protocol shows where the threshold $1/3$ comes from. Consider the case where the threshold is 1/2 and $2\ell+1 = L$. For a target client, the (malicious) server can tell $\ell/2$ decryptors that the client is online and tell another $\ell/2+1$ decryptors that the client is offline. Then combined with the ℓ malicious decryptors' shares, the server has $\ell/2+\ell$ shares to reconstruct individual mask, and $\ell/2+1+\ell$ shares to reeconstruct the pairwise mask.

Multi-round security. Our threat model assumes that $\mathcal{A}$ controls ηN clients throughout T rounds (§2.3). There are two things we need to additionally consider on top of the single-round proof: 1) the set S_t is generated from PRG, and 2) the pairwise mask $h_{i,j,t}$ computed from $\text{PRF}(r_{i,j}, t)$. For the former, we program $\mathcal{R}_{\text{PRG}}$ (like the single-round proof) such that the ChooseSet outputs S_t.

Now we analyze the per-round pairwise masks. Let the distribution of the view of $\mathcal{A}$ in round t be Δ_t. We next show that if there exists an adversary $\mathcal{B}$, and two round number $t_1, t_2 \in [T]$ such that $\mathcal{B}$ can distinguish between Δ_{t_1} and Δ_{t_2}, then we can construct an adversary $\mathcal{B}'$ who can break PRF security. We call the challenger in PRF security game simply as challenger. There are two worlds (specified by $b = 0$ or 1) for the PRF game. When $b = 0$, the challenger uses a random function; when $b = 1$, the challenger uses PRF and a random key for the PRF. We construct $\mathcal{B}'$ as follows. On input t_1, t_2 from $\mathcal{B}$, $\mathcal{B}'$ asks challenger for h_{i,j,t_1} for all clients i and j, and round t_1, t_2. Then $\mathcal{B}'$ creates the messages computed from $h_{i,j,t}$'s as protocol Π_{sum} prescribed;

Functionality $\mathcal{F}_{\text{mal-robust}}$

Parties: clients $1,\ldots,N$ and a server.

Parameters: corrupted rate η, dropout rate δ.

- $\mathcal{F}_{\text{mal-robust}}$ receives from the adversary $\mathcal{A}$ a set of corrupted parties, denoted as $\mathcal{C} \subset [N]$, where $|\mathcal{C}|/N \leq \eta$.
- For each round $t \in [T]$:

1) $\mathcal{F}_{\text{mal-robust}}$ receives a random subset $S_t \subset [N]$, a set of dropout clients $\mathcal{O}_t \subset S_t$, where $|\mathcal{O}_t|/|S_t| \leq \delta$ and $|\mathcal{C}|/|S_t| \leq \eta$, and inputs $\vec{x}_{i,t}$ for client $i \in S_t \backslash (\mathcal{O}_t \cup \mathcal{C})$.
2) $\mathcal{F}_{\text{mal-robust}}$ outputs $z_t = \sum_{i \in S_t \backslash (\mathcal{O}_t \cup \mathcal{C})} \vec{x}_{i,t}$.
   Depending on whether the server is corrupted by $\mathcal{A}$ or not, there are two cases:
   a) If the server is not corrupted, then $\mathcal{F}_{\text{mal-robust}}$ outputs z_t;
   b) If the server is corrupted, then $\mathcal{F}_{\text{mal-robust}}$ sends S_t to $\mathcal{A}$ and asks $\mathcal{A}$ for a set and whether it should continue or not: if $\mathcal{A}$ replies with a set $M_t \subseteq S_t$ such that $|M_t|/|S_t| \geq 1-\delta$, then $\mathcal{F}^*_{\text{mal}}$ outputs $z'_t = \sum_{i \in M_t \backslash \mathcal{C}} \vec{x}_{i,t}$; otherwise sends `abort` to all honest clients in S_t.

Figure 16: Ideal functionality for Flamingo with robustness.

it generates two views $\Delta_{t_1}, \Delta_{t_2}$ and sends to $\mathcal{B}$. $\mathcal{B}'$ outputs whatever $\mathcal{B}$ outputs.

Failure probability for T rounds. For a single round, we already showed that protocol $\Pi^{(t)}_{\text{sum}}$ securely realizes $\mathcal{F}^{(t)}_{\text{sum}}$ except probability $p = n \cdot 2^{-\kappa+1}$. The probability that for all the T rounds the protocol is secure is therefore $1-(1-p)^T$, which is approximately $1 - T \cdot p$ when $T \cdot p \ll 1$. Therefore, the probability of failure (there exists a round that fails the simulation) is $Tn2^{-\kappa+1}$.

Appendix E. Extension for Robustness

E.1. Robust protocol

As discussed in Section 9, we want the protocol to additionally guarantee robustness when the server is honest. Figure 16 gives the abstraction of robustness. Specifically, our robust version $\Pi_{\text{sum-robust}}$ has two key properties different from Π_{sum}: (1) client inputs are fixed after submitting their vectors in the report step; (i.e., malicious decryptors have no way to change the client inputs); (2) when the server is honest, the malicious clients cannot cause the protocol to abort, and the protocol will always output a sum of inputs from online clients. We treat clients who misbehave in the report step as equivalent to changing their inputs. We emphasize that this type of robustness is weaker than the robustness notion in other works [24,50], and is practically meaningful only if correct computation in the report step and input validation are ensured.

We now describe two key changes to Π_{sum} that together bring robustness. The full description of $\Pi_{\text{sum-robust}}$ is given as Figure 18, where we highlight the changes from Π_{sum} in Figure 13.

Using public-key encryption for both types of seeds. Instead of using symmetric encryption for the shares of $m_{i,t}$, each client encrypts $m_{i,t}$ directly (but not the shares) using ElGamal encryption. This reduces the number of ciphertexts appended to the masked vector compared to Π_{sum}; but more importantly, later we will see how it combines with another technique (zero-knowledge proof for discrete log) to bring robustness. See the details in the report step of Figure 18. This change in the report step will bring a change in the reconstruction step that the decryptors do threshold decryption on ciphertext $m_{i,t}$, instead of decrypting the shares of $m_{i,t}$ using symmetric keys.

Zero-knowledge proof in decryption. In the reconstruction step, in addition to decrypting, each decryptor u, for each ciphertext (c_0, c_1), proves to the server that $\log_{c_0}(c_0)^{s_u} = \log_g g^{s_u}$, where g^{s_u} is known to the server. Note that g^{s_u} can be generated in the DKG along the way in the setup phase and stored by the server.

To prove that $\log_{c_0}(c_0)^{s_u} = \log_g g^{s_u}$, each decryptor chooses a random $\beta \in \mathbb{Z}_q$ and sends to the server c_0^{β}, g^{β}. The server then sends to the decryptor a challenge $e \in \mathbb{Z}_q$. The decryptor computes $z = s_u \cdot e + \beta$ and sends z to the server. The sever checks that $(c_0^{s_u})^e \cdot (c_0)^{\beta} = c_0^z$ and $(g^{s_u})^e \cdot g^{\beta} = g^z$. We use Fiat-Shamir transform to make the proof non-interactive.

Due to this proof of decryption, the server can exclude bogus partial decryptions from malicious decryptors; since the sharing polynomial for secret key s is of degree ℓ and there are at least $2\ell+1$ honest online decryptors, the server is always able to reconstruct the result.

E.2. Proof of the robust protocol

Let Ψ_T be the sequential execution of Π_{setup} (Fig.11) and the T-round $\Pi_{\text{sum-robust}}$.

Theorem 4 (Security of Flamingo extension)**.** Let the security parameter be κ. Let $\delta, \delta_D, \eta, \eta_D$ be threat model parameters as defined (§5.1,§5.2). Let ϵ be the graph generation parameter (Fig.1). Let n be the number of clients in summation in each round. Assuming the existence of a PKI, a trusted source of initial randomness, a PRG, a PRF, an asymmetric encryption AsymEnc, a symmetric SymAuthEnc, and a signature scheme, if $2\delta_D + \eta_D < 1/3$ and $\epsilon \geq \epsilon^*$ (Lemma 1), then under the communication model defined in §2.2, protocol Ψ_T securely computes functionality $\mathcal{F}_{\text{mal-robust}}$ (Fig.16) in the presence of a malicious adversary controlling η fraction of the N clients (or including the server), except probability $Tn \cdot 2^{-\kappa+1}$.

Proof. We present our proof for the two cases: the server is corrupted by $\mathcal{A}$ or the server is honest.

When the server is corrupted. When the server is corrupted, we aim to show that the protocol can securely realize

Functionality $\mathcal{F}^{(t)}_{\text{sum-robust}}$

Parties: clients in S_t and a server.

Parameters: dropout rate δ and malicious rate η over $|S_t|$ clients.

- $\mathcal{F}^{(t)}_{\text{sum-robust}}$ receives a set $\mathcal{O}_t$ such that $|\mathcal{O}_t|/|S_t| \leq \delta$, and a set of corrupted parties, $\mathcal{C}$; and $\vec{x}_{i,t}$ from client $i \in S_t \backslash (\mathcal{C} \cup \mathcal{O}_t)$.

  Depending on whether the server is corrupted by the adversary $\mathcal{A}$ or not, there are two cases:

  1) If the server is not corrupted by $\mathcal{A}$, then $\mathcal{F}_{\text{mal}}$ outputs $z_t = \sum_{i \in S_t \backslash (\mathcal{C} \cup \mathcal{O}_t)} \vec{x}_{i,t}$;
  2) If the server is corrupted by $\mathcal{A}$, then $\mathcal{F}_{\text{mal}}$ asks $\mathcal{A}$ for a set: if $\mathcal{A}$ replies with a set $M_t \subseteq S_t$ such that $|M_t|/|S_t| \geq 1-\delta$, then $\mathcal{F}_{\text{mal}}$ outputs $z'_t = \sum_{i \in M_t \backslash \mathcal{C}} \vec{x}_{i,t}$; otherwise sends `abort` to all honest clients.

Figure 17: Ideal functionality for a single round t in the collection phase with robustness.

$\mathcal{F}^{(t)}_{\text{sum}}$ in the (b) case. Now $\mathcal{A}$ corrupts a set of clients (as the above case) and the server. The construction of simulator $\mathcal{S}$ in the ideal world and the following hyrbid arguments is the same as Appendix D.5.

When the server is not corrupted. We construct a simulator $\mathcal{S}$ in the ideal world that runs $\mathcal{A}$ as a subroutine.

1) $\mathcal{S}$ obtains z_t from $\mathcal{F}^{(t)}_{\text{sum-robust}}$ (Fig.17).
2) $\mathcal{S}$ received a set $\mathcal{O}_t$ from $\mathcal{R}_{\text{drop}}$ (see Appendix D.5).
3) (Report step) $\mathcal{S}$ computes offline/online labels from the set $\mathcal{O}_t$, and interact with $\mathcal{A}$ acting as the server. Here the collected masked inputs are all zeros and the seeds for the masks are generated by $\mathcal{S}$ itself.
4) (Cross-check step) $\mathcal{S}$ acts as honest decryptors in the cross-check step, and if all the honest decryptors would abort in the protocol prescription then $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and sends `abort` to $\mathcal{F}^{(t)}_{\text{sum}}$ and halts.
5) (Reconstruction step) $\mathcal{S}$ acts as the server in the reconstruction step, if the server would abort in the protocol prescription, then $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and sends `abort` to $\mathcal{F}^{(t)}_{\text{sum-robust}}$ and halts.

We construct a series of hybrid execution, starting from the real world to the ideal world execution.

Hybrid 1. The view of $\mathcal{A}$ in the real world execution is the same as the view of $\mathcal{A}$ in the ideal world when $\mathcal{S}$ would have the actual inputs of honest parties, $\{\vec{x}_i\}_{i \in S_t \backslash \mathcal{C}}$, the masks, and the shares of the secret key $SK = s$ ($\mathcal{S}$ in fact would know the s in full because of the threshold requirement $3\ell+1 = L$ and $2\delta_D + \eta_D < 1/3$).

Hybrid 2. $\mathcal{S}$ now instead of using the actual secret key s, it replaces s with 0 and sends the corresponding $|\mathcal{C} \cap \mathcal{D}| < L/3$ shares of 0 in $\mathbb{Z}_q$ to $\mathcal{A}$. This hybrid has identical distribution to the previous hybrid from the property of Shamir secret sharing.

Hybrid 3. $\mathcal{S}$ now instead of using the actual masks of honest parties, it replaces each mask with a uniformly random mask generated by $\mathcal{S}$ itself. This hybrid has identical distribution to the previous hybrid.

Hybrid 4. $\mathcal{S}$ now instead of using the actual inputs of honest parties $\{\vec{x}_i\}_{i \in S_t \backslash \mathcal{C}}$, it replaces each input vector with all zeros. Note that either the masked vectors not the vectors in the clear are seen by $\mathcal{A}$. This hybrid has identical distribution to the previous hybrid.

Hybrid 5. Same as the previous hybrid, except that the label messages from honest decryptors in the last step are replaced with the offline/online labels obtained from $\mathcal{R}_{\text{drop}}$. This hybrid is the ideal world execution, and the view of $\mathcal{A}$ in this hybrid is identical to the previous hybrid.

What left to show is that we need to show that in the simulation in the ideal world, $\mathcal{S}$ will not abort. In simulation step 4, from our threshold requirement $L = 3\ell + 1$ and Lemma 3, $\mathcal{S}$ will not abort because the server sends the same *req* to $\mathcal{D}$. In simulation step 5, $\mathcal{S}$ will only abort if more than $1/3$ of decryptors fail the zero-knowledge proof. This cannot happen because the $2\delta_D + \eta_D < 1/3$.

Combined with our analysis for the setup phase when the server is honest (Appendix B.2), we conclude that Ψ_T securely realizes $\mathcal{F}_{\text{mal-robust}}$ in the random oracle model. □

Comparison with BBGLR. When the server is honest, both BBGLR and Flamingo (without robustness) has no guaranteed output delivery; a malicious server in BBGLR can learn several subsets of sum (§2.3), while in Flamingo a malicious server learns a single sum from at least $1-\eta-\delta$ fraction of clients (§5, Fig.4). The robust version of Flamingo improves over Flamingo in the honest-server case: the output delivery is guaranteed (§E.2).

Another difference is the selection of clients per round, which is orthogonal to the single-round protocol design. A malicious server may select a set of clients (S_t) with many being malicious, and this violates the parameter constraints on η in BBGLR, and hence the server will learn the sum of inputs from a small number of honest clients; in contrast, in this setting, the server in Flamingo will obtain a random vector. However, the tradeoff (§10) is that BBGLR tolerates adaptive adversaries across rounds but Flamingo does not.

Collection phase with robustness: $\Pi_{\text{sum-robust}}$ for round t

Initial state from setup phase: each client $i \in [N]$ holds a value v and public key $PK = g^s$; each decryptor $u \in \mathcal{D}$ additionally holds a Shamir share of $SK = s$ (threshold ℓ with $3\ell + 1 = L$). The server has g^{s_u} for each decryptor $u \in \mathcal{D}$.

Parameters: $\delta_D + \eta_D < 1/3$.

1. **Report step.**

   Server performs the following:

   Compute a set $\mathcal{Q}_{graph} \leftarrow \text{CHOOSESET}(v, t, n_t, N)$ and a graph $G_t \leftarrow \text{GENGRAPH}(v, t, \mathcal{Q}_{graph})$; store $\{A_i(t)\}_{i \in \mathcal{Q}_{graph}}$ computed from G_t.

   Notify each client $i \in \mathcal{Q}_{graph}$ that collection round t begins.

   Each client $i \in \mathcal{Q}_{graph}$ performs the following:

   Compute $\mathcal{Q}^{local}_{graph} \leftarrow \text{CHOOSESET}(v, t, n_t, N)$, and if $i \notin \mathcal{Q}^{local}_{graph}$, ignore this round.

   Sample $m_{i,t} \xleftarrow{\$} \{0,1\}^\kappa$ and compute $\{h_{i,j,t}\}_{j \in A(i)} \leftarrow \text{PRF}(r_{ij}, t)$ for $j \in A(i)$, where r_{ij} is derived from PKI.

   Send to server a message $msg_{i,t}$ consisting of

   $$Vec_{i,t} = \vec{x}_{i,t} + \text{PRG}(m_{i,t}) + \textstyle\sum_{j \in A_t(i)} \pm\text{PRG}(h_{i,j,t}), \quad \text{AsymEnc}(PK, m_{i,t}), \quad \text{AsymEnc}(PK, h_{i,j,t}) \text{ for } j \in A_t(i)$$

   where $A_t(i) \leftarrow \text{FINDNEIGHBORS}(v, S_t, i)$, and AsymEnc is ElGamal encryption (Def.2), along with the signatures for each ciphertext.

2. **Cross check step.**

   Server performs the following:

   Denote the set of clients that respond within timeout as $\mathcal{Q}_{vec}$.

   Compute partial sum $\tilde{z}_t = \sum_{i \in \mathcal{Q}_{vec}} Vec_{i,t}$.

   Build decryption request req (req consists of clients in S_t to be labeled):

   for each $i \in \mathcal{Q}_{graph}$,

   if $i \in \mathcal{Q}_{vec}$, label i with "online",

   and attach $\text{AsymEnc}(PK, m_{i,t})$;

   else label i with "offline",

   and attach $\{\text{AsymEnc}(PK, h_{i,j,t})\}_{j \in A_t(i) \cap \mathcal{Q}_{vec}}$.

   Send to each $u \in \mathcal{D}$ the request req and a set of the attached ciphertexts $\mathcal{E}_i$ (in order to recover the mask(s) of client i).

   Each decryptor $u \in \mathcal{D}$ performs the following:

   Upon receiving a request req, compute $\sigma^*_u \leftarrow Sign(sk_u, req \| t)$, and send (req, σ^*_u) to all other decryptors via the server.

3. **Reconstruction step.**

   Each decryptor $u \in \mathcal{D}$ performs the following:

   Ignore messages with signatures (σ_i or σ^*_u) with round number other than t.

   Upon receiving a message (req, σ^*_u), run $b \leftarrow Vrf(pk_i, req, \sigma^*_u)$. Ignore the message if $b = 0$.

   Abort if u received less than $2\ell + 1$ same messages that were not ignored. Denote such message as req^*.

   For req^*, continue only if

   each client $i \in S_t$ is either labeled as "online" or "offline";

   the number of "online" clients is at least $(1 - \delta)n_t$;

   all the "online" clients are connected in the graph;

   each online client i has at least k online neighbors such that $\eta^k < 2^{-\kappa}$.

   For each $i \in \mathcal{Q}_{graph}$,

   For each $(c_0, c_1) \in \mathcal{E}_i$, send $c_0^{s_u}$, along with the zero-knowledge proof for $\log_g(g^{s_u}) = \log_{c_0}(c_0)^{s_u}$.

   Server completes the sum:

   Denote the set of decryptors whose messages have been received as U. Compute a set of interpolation coefficients $\{\beta_u\}_{u \in U}$ from U.

   For each $i \in \mathcal{Q}_{graph}$, verify the zero-knowledge proof and reconstruct the mask $m_{i,t}$ or $\{h_{i,j,t}\}_{j \in A_t(i) \cap \mathcal{Q}_{vec}}$:

   for each ElGamal ciphertext (c_0, c_1) of the mask, compute $c_1 \cdot (\prod_{u \in U} (c_0^{s_u})^{\beta_u})^{-1}$;

   Output $z_t = \tilde{z}_t - \text{PRG}(m_{i,t}) + \sum_{j \in A_t(i) \cap \mathcal{Q}_{vec}} \pm\text{PRG}(h_{i,j,t})$.

Figure 18: Collection protocol with robustness.