Heli: Heavy-Light Private Aggregation

Ryan Lehmkuhl
MIT

Henry Corrigan-Gibbs
MIT

Emma Dauterman
Stanford

David J. Wu
UT Austin

Abstract

This paper presents Heli, a system that lets a pair of servers collect aggregate statistics about private client-held data without learning anything more about any individual client's data. Like prior systems, Heli protects client privacy against a malicious server, protects correctness against misbehaving clients, and supports common statistical functions: average, variance, and more. Heli's innovation is that only *one* of the servers (the "heavy server") needs to do per-run work proportional to the number of clients; the other server (the "light server") does work sublinear in the number of clients, after a one-time setup phase. As a result, a computationally limited party, such as a low-budget non-profit, could potentially serve as the second server for a Heli deployment with millions of clients.

Heli relies on a new cryptographic primitive, *aggregation-only encryption*, that allows computing certain restricted functions on many clients' encrypted data. In a deployment with ten million clients, in which the servers privately compute the sum of 32 client-held 1-bit integers, Heli's heavy server does 240,000 core-s of work and the light server does 7 core-ms of work. Compared with prior work, the heavy server does 38× more computation, but the light server does 120,000× less.

1 Introduction

A private-aggregation system allows an app developer or hardware vendor to collect aggregate telemetry information from their users' devices, without needing to gather any sensitive disaggregated user data. Private-aggregation systems have been at the heart of privacy-friendly telemetry applications at Apple [7], Google [68], and Mozilla [88].

At the same time, private-aggregation technology is not in widespread use today—even privacy-focused products, such as encrypted-messaging apps, do not often use it. Well-resourced tech giants are responsible for the vast majority of successful deployments.

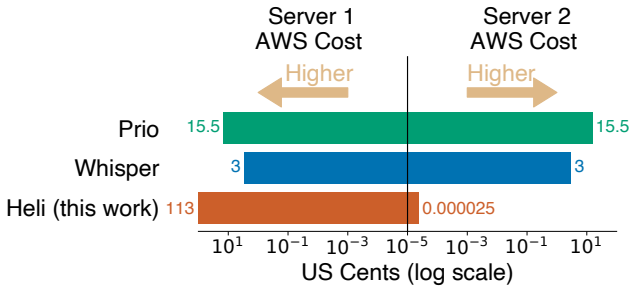

Figure 1: After a one-time setup, Heli reduces the AWS cost of aggregation for the second server by five orders of magnitude compared to Prio [45] and Whisper [94], but increases the first server's cost by up to 38×. The plot shows the current AWS cost in US cents to aggregate 32 boolean values over 10 million clients when 10% of clients don't participate.

One reason for the lack of use is that private-aggregation systems require at least two independent infrastructure providers, with security holding against an adversary that does not control both. (It is possible in principle to offload one provider's computation work onto clients [14, 15, 16, 24, 42, 72, 74, 80, 81, 83, 84, 85, 98], though this may be even more challenging to do in practice. We discuss these schemes in Section 7.)

A major drawback is that the second party needs to do work *linear* in the number of participating clients. Thus, if the second party is providing this service for thousands of applications, it must run server-side infrastructure to handle thousands of applications' worth of client requests. As a result, the one organization that does this work today, Divvi Up [51], charges money for the service and requires a business negotiation to use—both severe barriers to casual use [2].

Our goal is to drive down the cost of running the second private-aggregation party to the point that a single organization can afford to provide private-aggregation infrastructure for thousands of applications at once, as a free service.

As a step towards this goal, this paper presents Heli, a two-party private-aggregation scheme in which the second party's

work is *sublinear in the number of clients*, after a one-time setup. More specifically, in Heli, one of the parties (the "heavy server") does work proportional to the number of clients while the other party (the "light server") does much less—as little as constant per measurement, after a one-time setup. We envision a deployment in which the application provider runs the heavy server, since the application provider is likely doing work linear in the number of users of their service already. Then, a reputable but compute-limited provider, such as the ISRG [1], can operate the light server.

The core technical challenge in Heli is upholding the strong privacy guarantees of a traditional private-aggregation system without requiring the light server to communicate with every client for every measurement. To show why these two properties are in tension, consider the following straw man: to compute some statistic, each client encrypts its value under the light server's public key using an additively homomorphic encryption scheme [95] and sends the ciphertext to the heavy server. The heavy server sums the ciphertexts and sends the sum to the light server for decryption. While this approach requires the light server to decrypt just one ciphertext, a malicious adversary that compromises the heavy server can break privacy. For example, such an adversary could send a single client's submission to the light server to decrypt or compute a different function of user inputs than the sum. Security is further complicated by the fact that we want to collect measurements over many rounds; a malicious heavy server should not be able to mix ciphertexts across rounds either.

We address these challenges with a new cryptographic primitive that we call *aggregation-only encryption*. With aggregation-only encryption, it is possible for two parties to compute an aggregate statistic over many encrypted user submissions while preserving user privacy in the presence of a malicious adversary that can compromise one of the two parties. At a high level, aggregation-only encryption enables many clients to submit ciphertexts to an *aggregator* (the heavy server), which collects them into a single, compact ciphertext. The aggregator can then send the ciphertext to the *decryptor* (the light server), which can decrypt the aggregate ciphertext and validate that every client's contribution was included in the final result. We build our construction using a Diffie-Hellman-based key-homomorphic PRF. Our construction can be viewed as an extreme special case of private aggregation schemes such as LERNA [80] and OPA [74] that have committees of clients run the second party as an MPC. Collapsing the committee size of these schemes to one and applying some additional modifications yields an aggregation-only encryption scheme.

With aggregation-only encryption as the cryptographic core, we show how to construct a private-aggregation system called Heli. Most of the systems-level engineering is to handle malicious and unreliable clients. Heli is able to detect and exclude clients that deviate from the protocol, make progress in spite of clients that go offline, and allow clients to join the system after the initial setup phase.

We implement Heli and evaluate its performance across a range of benchmarks. In an end-to-end deployment that privately aggregates 32 binary values across ten million clients (with 10% of clients dropping out each round) aggregation costs 113 US-cents for the aggregator and 0.000025 US-cents for the decryptor; Heli's aggregator is at most 38× more expensive to operate than a Prio or Whisper server, while the decryptor is at least 120,000× cheaper. Figure 1 shows Heli's asymmetric server costs relative to Prio and Whisper.

Limitations. One important limitation is that the heavy server's cost in Heli becomes larger than conventional private-aggregation servers [45, 94] as the number or bitwidth of measurements grows. Both Heli and prior work use zero-knowledge proofs to defend against misbehaving clients; in our heavy/light setting, we cannot use the lightweight information-theoretic zero-knowledge techniques that prior systems use. A second limitation is that the light server must (1) run a one-time setup protocol with each new client and (2) do work per measurement that scales linearly with the number of *offline* clients.

Notation. For a natural number n, we write $[n] = \{1, 2, \ldots, n\}$. For sets $\mathcal{X}$ and $\mathcal{Y}$, $\mathsf{Funs}[\mathcal{X}, \mathcal{Y}]$ is the set of all functions from $\mathcal{X}$ to $\mathcal{Y}$, and $2^{\mathcal{X}}$ is the powerset of $\mathcal{X}$. "Efficient" means probabilistic polynomial time. For a finite set S, we use $x \xleftarrow{R} S$ to denote a uniform random sample from S. We use λ to denote the security parameter.

2 System overview

Heli allows a company, such as the developer of a mobile app, to collect telemetry data from their clients (e.g., app users) without seeing the clients' sensitive data in the clear.

Servers. A Heli deployment involves two servers, which we call the "aggregator" (i.e., the heavy server) and the "decryptor" (the light server). The system protects user privacy against a malicious adversary compromising one of the two servers and any number of clients.

The aggregator does the bulk of the computational work during the collection of each statistic. We envision the app developer as running this server, possibly outsourcing the computational work to a cloud provider. The decryptor communicates with each client once in a registration step; after this per-client setup, the servers can compute an unbounded number of statistics with no interaction between the clients and the decryptor. We envision a reputable but compute-limited infrastructure provider, such as the ISRG [1], as running the decryptor.

We assume that all parties in the system hold a copy of the two servers' long-term public keys.

Clients. Before using Heli, the servers must agree upon a strategy for client authentication—a technique for mapping each connecting client to some globally unique identifier,

which could be a public key, Google account name, etc. The servers must also agree upon a policy for which clients to admit to the system. For example, the servers could use Apple's identity infrastructure [9] to only admit genuine iPhone clients. In an open Internet setting, the servers might authenticate users via their SSH public keys and would admit any user with a valid public key.

At a certain point, the servers must agree that the set of client identities is "large enough" to proceed with aggregation. For example, the servers might agree to proceed whenever there are more than 10,000 valid Apple devices participating.

Tolerance to offline clients. In a large-scale deployment, some participating clients will go offline for periods of time. In any system (including ours) in which clients only communicate with one server during aggregation, there is an inherent trade-off between security and tolerance to offline clients: if the servers agree to make progress if up to a threshold t clients are offline, then a malicious aggregator can always exclude t clients' submission from the final sum without detection. During setup, the servers must agree on some policy to determine, given a candidate set of participating clients in a protocol round, whether to proceed with aggregation.

Aggregation function. The system is parameterized by a modulus p, a bound B, a function $f(x_1, \ldots, x_n) = \sum_{i \in [n]} a_i x_i$ with all $a_i \in \{1, \ldots, B\} \in \mathbb{Z}_p$. For each client i holding a value $m_i \in \{0, \ldots, B\}$ Heli allows the servers to compute the function $f(m_1, \ldots, m_n)$ without learning anything else. Further, the servers agree during setup on a predicate $\mathsf{Valid}\colon \mathbb{Z}_p \to \{0,1\}$ indicating which client-provided data values are "allowed;" each client's data value must satisfy $\mathsf{Valid}(m_i) = 1$. For example, if each client is a car and the manufacturer wants to gather average speeds, the predicate Valid might return '1' on inputs $\{0, 1, \ldots, 160\}$. (For simplicity, we focus on single-argument Valid predicates. Our approach naturally extends to multi-variate predicates, e.g., where clients upload (x, x^2) and prove that x is in a valid range and x^2 is correctly computed.) We focus on private sums since (1) many telemetry applications use only private sums, and (2) off-the-shelf techniques show how to use private sums to compute other simple statistics [45, 73, 86, 92].

2.1 Protocol flow

We describe the high-level flow of a Heli deployment below.

One-time client registration. When a new client with identity id joins the system, the decryptor generates a secret "encryption key" for the client and sends it to the client. The client never needs to communicate with the decryptor again. Once the decryptor has registered a sufficiently large set of clients $\mathcal{C} = \{\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_n\}$, the decryptor sends the list of client identities $\mathcal{C}$ to the aggregator.

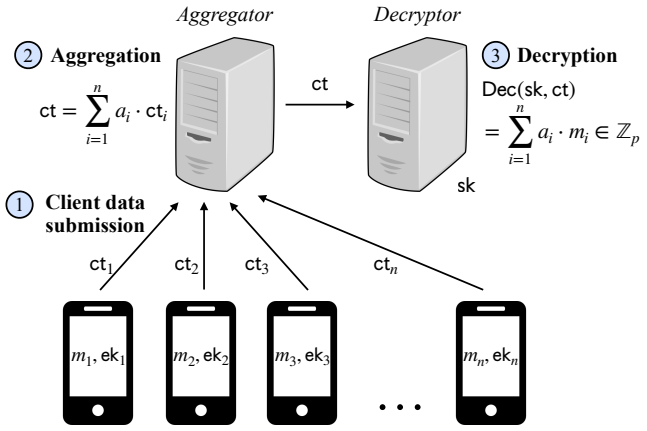

Figure 2: Heli protocol flow for one round of measurement collection. (1) Client i uses its encryption key ek_i, to encrypt its data value x_i, generating ct_i. (2) The aggregator sums the client ciphertexts to product the aggregate ciphertext ct. (3) The decryptor uses its secret key sk to obtain the result and check for aggregator misbehavior.

After registration, the servers can collect an unlimited number of aggregate statistics, in a sequence of data-collection protocol "rounds." The application assigns a unique per-round identifier string to each round. An unlimited number of protocol rounds can safely execute concurrently with clients using the same encryption keys across rounds. In each round, the clients communicate only with the aggregator.

The per-round protocol flow, depicted in Figure 2, is:

Step 1: Client data submission. Each client, with identity id_i, uses their secret encryption key (obtained from the decryptor during setup) along with the round identifier to encrypt their data value m_i for the round into a ciphertext ct_i. The client forwards this ciphertext ct_i to the aggregator.

Step 2: Aggregation (at heavy server). The aggregator combines the ciphertexts it received into a single ciphertext ct, whose size is independent of the number of clients. The aggregator forwards this ciphertext to the decryptor. (In Section 5, we explain how Heli handles offline clients.)

Step 3: Decryption (at light server). Finally, the decryptor receives the aggregate ciphertext ct from the aggregator. The decryptor attempts to decrypt the ciphertext using its secret key to recover the aggregate statistic. Decryption will only succeed if the aggregator behaved correctly. The decryptor can return this statistic to the aggregator, broadcast it to the clients, or otherwise publish it, depending on the needs of the application. The decryptor's per-round work is *sublinear* in the total number of clients in the system, though it does depend linearly on the number offline clients, as we explain in Section 5.

2.2 Security goals

Client-input privacy against one malicious server. Heli protects the privacy of honest clients' inputs against an attacker that compromises one of the servers and an arbitrary number of clients, even if these parties deviate from the protocol arbitrarily. A malicious server's influence is limited to its ability to (1) choose the inputs of corrupt clients, and (2) exclude a subset of honest clients, within limits determined by the system's configuration. Against a malicious server, Heli makes no guarantee about the correctness of the system's output.

Security against malicious clients. In addition, Heli provides full malicious security against a coalition of malicious clients, provided that the servers execute the protocol correctly. More precisely, the system parameters include a predicate $\mathsf{Valid}()$; Heli allows clients to submit data values m such that $\mathsf{Valid}(m) = 1$. The only influence that a malicious client has on the output is the influence allowed via its choice of a data value in the set $\{m \mid \mathsf{Valid}(m) = 1\}$.

To make these definitions more precise, we define an ideal functionality $\mathcal{F}_{\mathsf{Heli}}$ in Figure 3. A "set of inputs" I for an execution is, for each participant, the messages they submit to the ideal functionality, and a timestamp at which they submit.

Definition 2.1 (Client-input privacy against one malicious server)**.** For all efficient adversaries $\mathcal{A}$ controlling a subset of the clients and either the aggregator or the decryptor, there exists an efficient simulator $\mathcal{S}$ such that for all inputs I, the following distributions are computationally indistinguishable:

- *Real world:* The adversary $\mathcal{A}$'s view in a Heli protocol execution on inputs I.
- *Ideal world:* The output of the simulator $\mathcal{S}$, playing the role of the adversary, in an execution of $\mathcal{F}_{\mathsf{Heli}}$ on inputs I.

Definition 2.2 (Security against malicious clients)**.** For all efficient adversaries $\mathcal{A}$ controlling a subset of the clients, there exists an efficient simulator $\mathcal{S}$ such that for all inputs I, the following distributions are computationally indistinguishable:

- *Real world:* The joint distribution of (a) the adversary $\mathcal{A}$'s view and (b) the decryptor's output, in a Heli protocol execution on inputs I.
- *Ideal world:* The joint distribution of (a) the output of the simulator $\mathcal{S}$, playing the role of the adversary, and (b) the set of messages the decryptor receives, in an execution of $\mathcal{F}_{\mathsf{Heli}}$ on inputs I.

Ideal functionality $\mathcal{F}_{\mathsf{Heli}}$. Participants are: n clients, two servers (the aggregator and decryptor), and the adversary. The functionality is parameterized by:

(i) a message space $\mathcal{M}$, with default value $m^* \in \mathcal{M}$,
(ii) an aggregation function $f\colon \mathcal{M}^n \to \mathcal{M}$,
(iii) an input-validation predicate $\mathsf{Valid}\colon \mathcal{M} \to \{0,1\}$, which checks if a client's input is "allowed" and
(iv) a predicate $\mathsf{Proceed}\colon 2^{[n]} \to \{0,1\}$ that takes as input a set of client identifiers and indicates whether to reveal an aggregate statistic computed over these clients' inputs, and
(v) a number of time periods $T \in \mathbb{Z}^{\geq 0}$.

Initialization. For each time period $t \in [T]$, the functionality maintains the following state:

- $\mathsf{Inputs}_t \in \mathcal{M}^n$, each entry initialized to m^*,
- $\mathsf{Online}_t \subseteq [n]$, initialized to the empty set, and
- a time-period counter $t_{\mathsf{now}} \leftarrow 1$.

Client data submission. Each client $i \in [n]$ submits inputs of the form: $(t, m) \in [T] \times \mathcal{M}$. Upon receiving a client input, the ideal functionality proceeds as follows:

- Send a message (input, t, i) to the aggregator.
- If the aggregator or the decryptor is corrupted, set $v = 1$. Otherwise set $v = \mathsf{Valid}(m)$.
- If $t \geq t_{\mathsf{now}}$, $i \notin \mathsf{Online}_t$, and $v = 1$:
  - Set $\mathsf{Online}_t \leftarrow \mathsf{Online}_t \cup \{i\}$ and $\mathsf{Inputs}_t[i] \leftarrow m$.

Aggregation. The aggregator submits a set $U \subseteq [n]$ to the ideal functionality. The ideal functionality then responds as follows:

- If $t_{\mathsf{now}} > T$ or $U \not\subseteq \mathsf{Online}_{t_{\mathsf{now}}}$, respond with $\perp$ to the aggregator.
- For $i \in [n]$, set $m_i \leftarrow \mathsf{Inputs}_{t_{\mathsf{now}}}[i]$.
- If $\mathsf{Proceed}(U) = 1$, set $y \leftarrow f(m_1, \ldots, m_n)$.
- Otherwise, set $y \leftarrow \perp$.
- Send $(\mathsf{output}, t_{\mathsf{now}}, U, y)$ to the aggregator and decryptor and set $t_{\mathsf{now}} \leftarrow t_{\mathsf{now}} + 1$.

Figure 3: Ideal functionality of Heli.

2.3 Homomorphic encryption is not enough

The overview of Heli in Section 2.1 explains that Heli relies on the aggregator's ability to aggregate many client ciphertexts into a single ciphertext. The question then arises: what encryption scheme should we plug in to this framework? The right answer, as we will show, is aggregation-only encryption, a new type of encryption scheme that we describe in Section 4.

First, we explain why the system would be *insecure* if we instead used a vanilla homomorphic-encryption scheme [95]. Two problems arise when using homomorphic encryption in our setting.

The first problem with this approach is that a malicious aggregator could learn more than it should about honest clients' data. For example, a malicious aggregator could set the aggregate ciphertext ct equal to client 1's ciphertext $\mathsf{ct} = \mathsf{ct}_1$. The decryptor would not be able to detect this attack and the aggregate statistic could leak client 1's input data in the clear. Somehow we need to ensure that decryption fails whenever a malicious aggregator produces the "wrong" aggregate ciphertext.

A related problem is domain separation across protocol rounds: If the parties want to run the data collection many times, they need some way to ensure that a malicious aggregator cannot replay ciphertexts from round t in round $t+1$ in a way that causes a privacy violation. Even if the parties used fresh encryption keys in each round, subtle cross-round attacks seem hard to prevent [52].

3 Background

Short-interval discrete logarithm. We will need to compute small discrete logarithms in cryptographic groups. In particular, for a cyclic group $\mathbb{G}$ with generator g and integer M such that $1 \le M \le |\mathbb{G}|$, we define the M-bounded discrete-log decoding procedure $\mathsf{DLog}_M : \mathbb{G} \to [M] \cup \{\perp\}$ as:

$$\mathsf{DLog}_M(y) = \begin{cases} m & \text{if } y = g^m \text{ for } m \in [M], \\ \perp & \text{otherwise.} \end{cases}$$

An implementation of DLog_M using the baby-step giant-step algorithm runs in time roughly $\sqrt{M}$, or $M^{1/3}$ with preprocessing [46]. In our applications, the bound M is always polylog($|\mathbb{G}|$), which means that the algorithm is efficient.

Key-homomorphic PRFs. A pseudorandom function [65] (PRF) with key space $\mathcal{K}$, input space $\mathcal{X}$ and output space $\mathcal{Y}$ is an efficient deterministic function $F\colon \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$. A PRF is secure if, for a random key $k \xleftarrow{R} \mathcal{K}$, no efficient algorithm can distinguish $F(k,\cdot)$ from a truly random function. Moreover, a pseudorandom function F is *key homomorphic* [30] if the key space and output space are additive groups such that for all $k_0, k_1 \in \mathcal{K}$ and all $x \in \mathcal{X}$, it holds that $F(k_0,x) + F(k_1,x) = F(k_0 + k_1, x)$.

The following is due to Naor, Pinkas, and Reingold [89]:

Construction 3.1 (Key-homomorphic PRF [89])**.** Let $\mathbb{G}$ be a cyclic group of prime order p in which the decisional Diffie-Hellman (DDH) assumption holds [25]. Let $H\colon \{0,1\}^* \to \mathbb{G}$ be a hash function. Define $\mathcal{K} = \mathbb{Z}_p$, $\mathcal{X} = \{0,1\}^*$, and $\mathcal{Y} = \mathbb{G}$. Then the function: $F(k,x) := H(x)^k$ is a key-homomorphic PRF, provided that we model H as a random oracle.

4 New tool: Aggregation-only encryption

In this section, we introduce aggregation-only encryption, the new cryptographic primitive at the core of Heli. An aggregation-only encryption scheme for a function f allows n independent (and mutually distrusting) clients, each holding an encryption key, to encrypt messages $m_1, \ldots, m_n$. An *aggregator* can then aggregate the n client ciphertexts into a single, short aggregate ciphertext. A *decryptor*, who holds a secret decryption key, can decrypt the aggregated ciphertext, revealing the plaintext $f(m_1, \ldots, m_n)$.

The aggregator can also specify a subset $D \subseteq [n]$ of clients that dropped out (i.e., did not contribute a ciphertext). If the decryptor runs decryption with respect to this same dropped-out set D, the revealed plaintext is instead $f(m'_1, \ldots, m'_n)$ where $m'_i = m_i$ for all $i \notin D$, and $m'_i = 0$ (or some other default value) for all $i \in D$.

We require that an aggregation-only encryption scheme reveal nothing more about the honest clients' messages than the output of the aggregation function, applied honestly to all n clients' messages (where the message for dropped-out clients $D \subseteq [n]$ is replaced by the default message m^*), reveals. We require this property to hold against an adversary that compromises:

- an arbitrary number of clients; and
- *either* (a) the aggregator or (b) the decryptor, who also generates the clients' encryption keys.

Moreover, we also require security to hold if the parties execute this aggregation protocol over multiple rounds using the same set of keys (but with distinct round-identifier strings).

Technical challenge. The challenge in constructing aggregation-only encryption is ensuring that, for client messages $m_1, \ldots, m_n$, neither the aggregator nor the decryptor can deviate from the protocol to learn more than $f(m_1, \ldots, m_n)$. For example, the aggregator should not be able to aggregate just one client's ciphertext, learn the output of a different function f', or mix-and-match ciphertexts across rounds.

Construction overview. We build an aggregation-only encryption system for computing linear functions (with small coefficients) over private client data. Our construction makes black-box use of a key-homomorphic pseudorandom function (Section 3). When using a Diffie-Hellman-based key-homomorphic PRF (Construction 3.1), generating a submission requires the client to perform two group exponentiations, aggregating n submissions requires one exponentiation per-client (or one group operation per-client if computing a simple sum), and decryption requires 1 group exponentiation and a small-interval discrete-log computation (see Section 5.5). Each client's encrypted submission is 1 group element.

Our construction can be viewed as an extreme special case of prior single-server aggregation protocols LERNA [80] and OPA [74]. These schemes sample a subset of clients to form a "decryption committee" that ensures a malicious aggregator learns nothing more than the honest aggregation output. Shrinking the committee size of these schemes to one (in the right way) and applying some additional preprocessing produces our aggregation-only encryption scheme.

4.1 Syntax

An aggregation-only encryption scheme is parameterized by a message space $\mathcal{M}$, a round-identifier space $\mathcal{R}$, and a function class $\mathcal{F} \subseteq \mathsf{Funs}[\mathcal{M}^*, \mathcal{M}]$. It consists of four efficient algorithms:

- $\mathsf{Gen}(1^\lambda, f) \to (\mathsf{sk}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$. Given a security parameter λ and a function $f \in \mathcal{F}$ of type $f\colon \mathcal{M}^n \to \mathcal{M}$, output a secret decryption key sk and (secret) encryption keys $\mathsf{ek}_1, \ldots, \mathsf{ek}_n$.
- $\mathsf{Enc}(\mathsf{ek}_i, r, m) \to \mathsf{ct}_i$. Given secret encryption key ek_i, round identifier $r \in \mathcal{R}$, and message $m \in \mathcal{M}$, output ciphertext ct_i.
- $\mathsf{Apply}(f, D, \{\mathsf{ct}_i\}_{i \in [n] \setminus D}) \to \mathsf{ct}$. Given a function $f \in \mathcal{F}$ of type $f\colon \mathcal{M}^n \to \mathcal{M}$, a drop-out set $D \subseteq [n]$, and a collection of ciphertexts ct_i for $i \in [n] \setminus \mathcal{D}$, output an aggregated ciphertext ct.
- $\mathsf{Dec}(\mathsf{sk}, r, D, \mathsf{ct}) \to m \in \mathcal{M}$ or $\perp$. Given a secret decryption key sk, round identifier $r \in \mathcal{R}$, a drop-out set D, and a ciphertext ct, output a message m or the failure symbol $\perp$.

4.2 Definition of aggregation-only encryption

We now describe the properties that an aggregation-only encryption scheme must satisfy. We define these formally in Appendix A.

Correctness. An aggregation-only encryption scheme is *correct* if, for any drop-out set $D \subseteq [n]$, encrypting messages m_i for $i \in [n] \setminus D$ under the same round identifier, running $\mathsf{Apply}(f, D, \cdot)$ on the resulting ciphertexts, and decrypting under the same round identifier and drop-out set D, yields $f(m_1, \ldots, m_n)$, where the message associated with dropped-out clients $i \in D$ is $m_i = m^*$. In this work, $m^* = 0$.

Compactness. An aggregation-only encryption scheme $\mathcal{E}$ is *compact* if the running time of the decryption algorithm is a fixed polynomial in the security parameter λ and the size of the drop-out set D, and is sublinear in the arity n of the function f passed to Gen. Homomorphic encryption schemes require a similar form of compactness [64]. This requirement rules out trivial schemes where the Apply function just concatenates its arguments.

In our application, this requirement ensures that the decryptor's running time is linear in the number of dropped-out clients, and is sublinear in the total number of clients.

Security. We demand security against an adversary that compromises an arbitrary subset of the clients and *either* the aggregator (who runs the Apply algorithm) or the decryptor (who runs the Gen and Dec algorithms), and that runs the aggregation scheme with the same keys over many rounds. Against such an adversary, the encryption scheme should reveal nothing more about honest clients' inputs than what the adversary would learn by seeing the aggregation function f applied to the online clients' inputs in each round.

We give a precise simulation-based definition in Appendix A. A few comments about our definition:

Clients can safely submit ciphertexts for multiple rounds at once. In some applications, it may make sense to run multiple rounds of our aggregation scheme in parallel. Our security definition says that such parallel execution is safe. In particular, when defining security against a malicious aggregator, we allow the aggregator to see all honest clients' ciphertexts for all rounds at once at the start of the experiment. (In contrast, the malicious decryptor never sees unaggregated ciphertexts.) Even in such a setting, a malicious aggregator cannot learn more than it should about honest clients' inputs.

Security holds even against maliciously generated keys. When defining security against a malicious decryptor, we allow the adversary to generate all of the clients' keys. This implies that it is safe in a deployment to have the (potentially malicious) decryptor run key generation and distribute keys to clients.

The basic scheme does not protect correctness *against malicious clients or servers.* Our basic aggregation-only encryption scheme does not guarantee that the servers obtain the correct aggregate statistic if any party misbehaves. In Section 5, we discuss how to protect against malicious clients.

4.3 Construction

We present our construction of aggregation-only encryption for linear functions in Construction 4.1. The construction supports a deployment setting in which each client $i \in \{1, \ldots, n\}$, holds a (small) input $m_i \in \mathbb{Z}_p$, for a large prime modulus p. The scheme supports linear aggregation functions over $\mathbb{Z}_p$ with small coefficients.

More precisely, the construction is parameterized by a bound $B \in \mathbb{Z}^{\geq 0}$. The construction then supports all functions f, where $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_n \bmod p$, where all coefficients a_i are in $\{1, \ldots, B\}$, and all inputs x_i are in $\{0, \ldots, B\}$.

Construction idea. We build our construction from a key-homomorphic PRF (Section 3).

The high-level idea is to have each client generate a per-round mask using their encryption key that they use to encode their message. When the aggregator sums together ciphertexts, it also combines the client masks. Then at decryption time, the decryptor uses its secret key to remove client masks from the aggregate ciphertext.

Let the aggregation function be $f(x_1, \ldots, x_n) = \sum_{i \in [n]} a_i x_i \in \mathbb{Z}_p$. In the following, we assume that the coefficients $a_1, \ldots, a_n$ as well as the client inputs $x_1, \ldots, x_n$ are drawn from a fixed-size range $\{0, \ldots, B\}$. The construction works as follows:

Key generation. The decryptor samples random keys $(\mathsf{ek}_1, \ldots, \mathsf{ek}_n)$ for a key-homomorphic PRF with key-space $\mathbb{Z}_p$. The de-

cryptor computes the aggregated key $k = \sum_{i\in[n]} a_i \cdot \mathsf{ek}_i \in \mathbb{Z}_p$ and stores the PRF keys $(k, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$. The decryptor sends key ek_i to client $i \in [n]$.

Client data submission. To encrypt their message $m_i \in \{0, \ldots, B\} \subseteq \mathbb{Z}_p$ in a given round, client i first uses its secret key ek_i and the round-identifier string $r \in \{0,1\}^\lambda$ to derive a per-round mask $R_i \leftarrow F(\mathsf{ek}_i, r) \in \mathbb{G}$. Then the client's ciphertext ct_i, which it sends to the aggregator, is simply their input multiplied by the mask, $\mathsf{ct}_i = R_i \cdot g^{m_i}$.

Aggregation. Let $D \subseteq [n]$ be the subset of clients that drop out. The aggregator combines the ciphertexts from the online clients, $\mathsf{ct} \leftarrow \prod_{i\in[n]\setminus D} \mathsf{ct}_i^{a_i} \in \mathbb{G}$, and sends the aggregate ciphertext ct to the decryptor.

Decryption. If all the online clients and the aggregator behaved honestly, the ciphertext $\mathsf{ct} = \prod_{i\in[n]\setminus D} g^{a_i \cdot m_i} \cdot F(\mathsf{ek}_i, r)^{a_i}$. By relying on key homomorphism of the PRF, the decryptor can compute the masking component by first computing

$$\begin{aligned} k' &= k - \textstyle\sum_{i\in D} a_i \cdot \mathsf{ek}_i \\ &= \textstyle\sum_{i\in[n]} a_i \cdot \mathsf{ek}_i - \sum_{i\in D} a_i \cdot \mathsf{ek}_i = \sum_{i\in[n]\setminus D} a_i \cdot \mathsf{ek}_i \end{aligned}$$

and then evaluating $R = F(k', r)$. The time required to compute the aggregate key only depends on the number of clients that drop out $|D|$, not on the total number of clients n. Finally, the decryptor computes the discrete-log of $\mathsf{ct}/R = g^{\sum_{i\in[n]\setminus D} a_i \cdot x_i}$ and publishes the result $m \in \mathbb{Z}_p$.

The maximum possible value of the discrete log of the output value y is $\sum_{i\in[n]} (\max_i m_i a_i) = nB^2$. The discrete-log calculation thus returns the correct output. (A malicious client could disrupt the scheme by submitting a large/malformed data value m_i. We discuss protection against such attacks in Section 5.)

Correctness. This follows by construction.

Compactness. The running time of decryption depends linearly on the number of dropped-out clients $|D|$ and sublinearly in the total number of clients n. This is so because the final discrete-log calculation takes time $\sqrt{nB^2}$, with B being either constant or a fixed polynomial in the security parameter.

Security analysis. We prove the security of our construction in the full version of the paper [79]. We summarize the security analysis here.

Informal Theorem (Summarizing Theorems B.3 and B.6 of [79])**.** *When instantiated with a secure key-homomorphic pseudorandom function (Section 3), our aggregation-only encryption construction for linear functions (Construction 4.1) is computationally secure against a malicious aggregator and is unconditionally secure against a malicious decryptor.*

Proof idea. Security against a malicious aggregator follows from the security of the key-homomorphic PRF. In particular, honest clients mask their input using the output of a PRF evaluated on a key that the aggregator doesn't know, so any linear combination of honest clients' ciphertexts other than the honest one will result in the honest decryptor outputting $\perp$ with overwhelming probability.

Security against a malicious decryptor follows from the fact that the aggregated ciphertext that the decryptor receives reveals nothing more than the sum of all online clients' contributions. □

Construction 4.1 (Our construction: Aggregation-only encryption for linear functions)**.** The construction is parameterized by a message-size bound $B \in \mathbb{Z}$ and a modulus p. The message space is $\mathbb{Z}_p$ and the round-identifier space is $\{0,1\}^\lambda$. The supported function class is the set of linear functions over $\mathbb{Z}_p$ with coefficients in $\{1, \ldots, B\}$. The default value for dropped-out clients is 0. The construction uses a key-homomorphic PRF $F\colon \mathcal{K} \times \{0,1\}^\lambda \to \mathbb{G}$ with key space $\mathcal{K}$, domain $\{0,1\}^\lambda$, and output space $\mathbb{G}$.

$\mathsf{Gen}(1^\lambda, f) \to (\mathsf{sk}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$.

- Write f as $f(x_1, \ldots, x_n) = \sum_{i\in[n]} a_i \cdot x_i \in \mathbb{Z}_p$.
- Sample $\mathsf{ek}_1, \ldots, \mathsf{ek}_n \xleftarrow{R} \mathcal{K}$.
- Compute $k \leftarrow \sum_{i\in[n]} a_i \cdot \mathsf{ek}_i \in \mathbb{Z}_p$
- Set $\mathsf{sk} \leftarrow (f, k, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$.
- Output $(\mathsf{sk}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$.

$\mathsf{Enc}(\mathsf{ek}_i, r, m) \to \mathsf{ct}$. *// Requires:* $m \in \{0, \ldots, B\}$.

- Compute $R_i \leftarrow F(\mathsf{ek}_i, r) \in \mathbb{G}$.
- Output $R_i \cdot g^m$.

$\mathsf{Apply}(f, D, \{\mathsf{ct}_i\}_{i\in[n]\setminus D}) \to \mathsf{ct}$.

- Write f as $f(x_1, \ldots, x_n) = \sum_{i\in[n]} a_i x_i \in \mathbb{Z}_p$.
- Output $\mathsf{ct} \leftarrow \prod_{i\in[n]\setminus D} \mathsf{ct}_i^{a_i} \in \mathbb{G}$.

$\mathsf{Dec}(\mathsf{sk}, r, D, \mathsf{ct}) \to m$ or $\perp$.

- Parse sk as a secret key $(f, k, \mathsf{ek}_1, \ldots, \mathsf{ek}_n)$.
- Write f as $f(x_1, \ldots, x_n) = \sum_{i\in[n]} a_i x_i \in \mathbb{Z}_p$.
- Compute $k' \leftarrow k - \sum_{i\in D} a_i \cdot \mathsf{ek}_i \in \mathbb{Z}_p$.
- Compute $y \leftarrow \mathsf{ct}/F(k', r) \in \mathbb{G}$.
- Output $m \leftarrow \mathsf{DLog}_{(n\cdot B^2)}(y) \in \mathbb{Z}_p$

4.4 Handling longer data vectors

In many applications of private-aggregation, each client holds a *vector* of data values in each round, and the servers want the sum of these vectors in each round. For example, if a browser vendor is collecting information on which of d

different browser features clients use, each client's data vector might be in $\{0,1\}^d$—one bit for each potential feature.

Our construction supports aggregation of ℓ metrics in a single round: a client can run the Enc routine many times with distinct round identifiers r and can send all of the ciphertexts to the aggregator in a batch.

4.5 Post-quantum security

The security of our aggregation-only encryption construction (Construction 4.1) relies on the security of the underlying key-homomorphic PRF, whose outputs are pseudorandom elements of the group used to encode messages. Following prior work on single-server aggregation [74, 80], we can instantiate this PRF using a lattice-based key-homomorphic construction over an appropriate group, yielding a candidate aggregation-only encryption scheme for linear functions (with small coefficients). We expect this simple construction to be post-quantum secure, though we leave a formal treatment of post-quantum security to future work.

One subtlety is that the lattice-based key-homomorphic PRFs (e.g., [30]) only satisfy approximate correctness where homomorphic operations may introduce some error in the low-order bits of their output. A standard way to handle the error is to have clients first pad their input values with a few zeros in the low-order bits and then round away the error at decryption time.

5 Heli system design

In this section, we describe how Heli uses aggregation-only encryption to build an end-to-end private-aggregation system. The additional challenge here is handling *unreliable clients*: they may deviate from the protocol, go offline between aggregation rounds, or join the system after setup.

5.1 System parameters

Heli uses our aggregation-only encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Apply}, \mathsf{Dec})$ of Construction 4.1. That scheme is parameterized by a bound $B \in \mathbb{Z}^{\geq 0}$ and a modulus p. To use the scheme, the servers must agree on the following public values:

- an aggregation function $f(x_1, \ldots, x_n)$; in the following description, we focus on linear aggregation functions $f(x_1, \ldots, x_n) = \sum_{i \in [n]} a_i x_i \in \mathbb{Z}_p$ with coefficients $a_i \in \{1, \ldots, B\}$,
- an input-validation predicate $\mathsf{Valid}\colon \mathbb{Z}_p \to \{0,1\}$ that checks if a client's input is "allowed" (at a minimum, Valid tests whether the client's input is in $\{0, \ldots, B\}$),
- a predicate $\mathsf{Proceed}\colon 2^{[n]} \to \{0,1\}$ that takes as input a set of client identifiers and indicates whether to reveal an aggregate statistic computed over these clients' inputs,
- a list of distinct round identifiers $r_1, \ldots, r_T \in \{0,1\}^*$, one per aggregation time period (these can be arbitrary—e.g., for a daily statistic these could be the encoding of each date in a year), and
- a mechanism for client authentication (e.g., public keys).

All parties also hold copies of the servers' public keys.

5.2 Protocol flow

We explain how Heli implements the protocol flow we overviewed in Section 2.1. All communication takes place over encrypted and authenticated channels established using server public keys and a client-authentication mechanism.

Initialization: Aggregator. For each aggregation time period $t \in [T]$, the aggregator maintains an array S_t of client-uploaded ciphertexts for round t. Each set begins empty. The aggregator also maintains a counter $t \in \mathbb{Z}^{\geq 0}$ of the current time period, initialized to $t \leftarrow 0$.

Initialization: Decryptor. The decryptor also maintains a counter $t \in \mathbb{Z}^{\geq 0}$ of the current time period, initialized to $t \leftarrow 0$.

The decryptor runs the key-generation algorithm Gen, deriving each per-client encryption key ek_i in Gen from a pseudorandom function. This allows the decryptor to easily regenerate offline clients' encryption keys without having to store them individually. Specifically, the decryptor samples a key k_{PRF} for a pseudorandom function $G\colon \mathcal{K} \times [n] \to \mathbb{Z}_p$. For all $i \in [n]$, the decryptor computes $\mathsf{ek}_i \leftarrow G(k_{\mathsf{PRF}}, i)$. It also computes the aggregate key $k \leftarrow \sum_{i \in [n]} a_i \cdot \mathsf{ek}_i \in \mathbb{Z}_p$.

Client registration. To join the system, the i^{th} client contacts and authenticates to the decryptor with their identity id. The decryptor validates the client's identity and sends $\mathsf{ek}_i \leftarrow G(k_{\mathsf{PRF}}, i)$ to the client. To the aggregator, the decryptor sends the index i of the client, its identity id, and a cryptographic commitment C_i to the client's encryption key ek_i: $C_{\mathsf{id}} \leftarrow \mathsf{Commit}(\mathsf{ek}_i)$. The aggregator will use this commitment in the next step to check the well-formedness of the client's submission.

Step 1: Client data submission. Client i holds an encryption key ek_i, a round identifier $r_t \in \{0,1\}^\lambda$ for the current time period t, and its data value $m_i \in \mathbb{Z}_p$ such that $\mathsf{Valid}(m_i) = 1$. (Recall from Section 5.1 that $\mathsf{Valid}(\cdot)$ tests whether a client input value is allowed.) The client then runs the encryption algorithm for our aggregation-only encryption scheme: $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{ek}_i, r_t, m_i)$.

The client generates a non-interactive zero-knowledge proof of knowledge π_i [23, 66] attesting to the fact that its ciphertext ct_i is well-formed. That is, the client proves knowledge of a data value $\widetilde{m}_i$ and an encryption key ek_i such that:

1. the submission is valid: $\mathsf{Valid}(\widetilde{m}_i) = 1$,
2. the client's ciphertext is consistent with these values: $\mathsf{ct}_i = \mathsf{Enc}(\widetilde{\mathsf{ek}}_i, r_t, m_i)$, where r_t is the round identifier for time period t, and

3. the client's encryption key $\widetilde{\mathsf{ek}_i}$ is consistent with the commitment that the aggregator holds: $C_i = \mathsf{Commit}(\widetilde{\mathsf{ek}_i})$.

We provide more detail on the proof in Section 5.4.

Step 2: Aggregation (at heavy server). The client sends the aggregator the tuple $(t, \mathsf{ct}_i, \pi_i)$. The aggregator checks the proof π_i against the client-submitted ciphertext ct_i and the decryptor-provided commitment C_i. If the proof fails, the aggregator rejects the submission. Otherwise, the aggregator adds the pair (i, ct_i) to the set S_t of ciphertexts for this time period.

The aggregator then checks if enough clients have submitted ciphertexts for the current time period t to allow decryption to proceed. Precisely, letting $i_1, i_2, \ldots$ denote the identities attached to the ciphertexts in set S_t, the aggregator computes the set of dropped-out clients $D \leftarrow [n] \setminus \{i_1, i_2, \ldots\}$. The aggregator then checks if $\mathsf{Proceed}([n] \setminus D) = 1$.

If so, the aggregator computes $\mathsf{ct} \leftarrow \mathsf{Apply}(f, D, \{\mathsf{ct}_i\}_{i \in [n] \setminus D})$, where f is the sum function, and ct_i is a ciphertext in set S_t. The aggregator forwards the tuple (D, ct) to the decryptor.

Step 3: Decryption (at light server). Upon receiving (D, ct) from the aggregator in time period t, the decryptor first checks that $\mathsf{Proceed}([n] \setminus D) = 1$. If $\mathsf{Proceed}([n] \setminus D) = 0$, the decryptor outputs $\perp$ as the statistic for time period t. Otherwise, it runs the aggregation-only encryption decryption algorithm: $m \leftarrow \mathsf{Dec}(\mathsf{sk}, r_t, D, \mathsf{ct})$. The decryptor outputs m as the statistic for time period t. In both cases, it then increments $t \leftarrow t + 1$.

5.3 Security analysis

We analyze the security of Heli in the full version of this paper [79]. To summarize, we show that any adversary that breaks input privacy by corrupting either server along with a subset of clients can be used to break the underlying aggregation-only encryption scheme. Additionally, we show that security against malicious clients follows from the knowledge-soundness of the zero-knowledge proofs.

5.4 Implementation note: ZK proofs

When the client sends its ciphertext to the aggregator, the client proves to the aggregator (in zero knowledge), that (1) the client's ciphertext is well-formed with respect to its encryption key and a round identifier, and (2) the client's encrypted message is well-formed with respect to the application's validity predicate.

To execute this proof, the system can use any discrete-log-based proof system [31, 32, 33]. Concretely, our construction (Construction 4.1) uses a group $\mathbb{G}$ of prime order p. We sketch how this works in Heli.

During client registration: To commit to the client's encryption key, $\mathsf{ek}_i \in \mathbb{Z}_p$, the decryptor sends to the aggregator a commitment $C_{\mathsf{id}} \leftarrow g_{\mathrm{com}}^{\mathsf{ek}_i} \in \mathbb{G}$ to the client's encryption key, for a public, random generator $g_{\mathrm{com}} \in \mathbb{G}$.

During data submission: The client uses a discrete-log proof of equality to prove that its aggregation-only encryption ciphertext encrypts the tuple $(r_t, \mathsf{ek}_i, m_i) \in \{0,1\}^* \times \mathbb{Z}_p^2$, where r_t is the round identifier for time period t, ek_i is their encryption key, and m_i is a value in $A \subseteq \mathbb{Z}_p$. (When the client's data is a vector, the construction generalizes naturally.)

Each client proves that their ciphertext is well-formed using a Schnorr proof [33, 97] and proves that their encrypted message is well-formed with Bulletproofs [32, 43]. We use two different proof systems since Schnorr proofs efficiently handle discrete-log relations, while Bulletproofs efficiently handle range constraints. We evaluate proof overheads in Section 6 and describe them in more detail in the full version of the paper [79].

5.5 Outsourcing decryptor work

Our aggregation-only encryption construction (Construction 4.1) requires the decryptor to solve a discrete-log instance to recover the final aggregation result. If the final sum is a b-bit value, the decryptor must compute $2^{b/2}$ group operations. When summing up one-bit values from a billion clients, this requires about 30k elliptic-curve operations—under a second of computation on a standard CPU. Using a precomputed table of size $2^{b/3} \cdot \mathsf{poly}(b)$, the decryptor can reduce the decoding time to $2^{b/3} \cdot \mathsf{poly}(b)$ operations [18, 46, 87]. Alternatively, or in addition, the decryptor can partially decrypt the aggregate ciphertext and outsource the final discrete-log computation to the aggregator. This outsourcing has no impact on security.

5.6 Extension: Differential privacy

The aggregate statistic itself may reveal information about individual clients' data. Like prior private aggregation schemes [4, 27, 45, 49, 93, 94], we can compose Heli with differential privacy techniques [54] to limit the information revealed by the final aggregate statistic. The aggregator can sample noise value ν_1 from a Laplace distribution, and add this value to the aggregate ciphertext that it sends to the decryptor. The decryptor decrypts, adds its own noise value ν_2 and publishes the resulting statistic. The output will be shifted by $\nu_1 + \nu_2$. If either the aggregator or decryptor is honest, and the noise is sampled from an appropriate distribution the resulting output distribution will be differentially private.

6 Evaluation

We evaluate the performance of Heli through several microbenchmarks and demonstrate that:

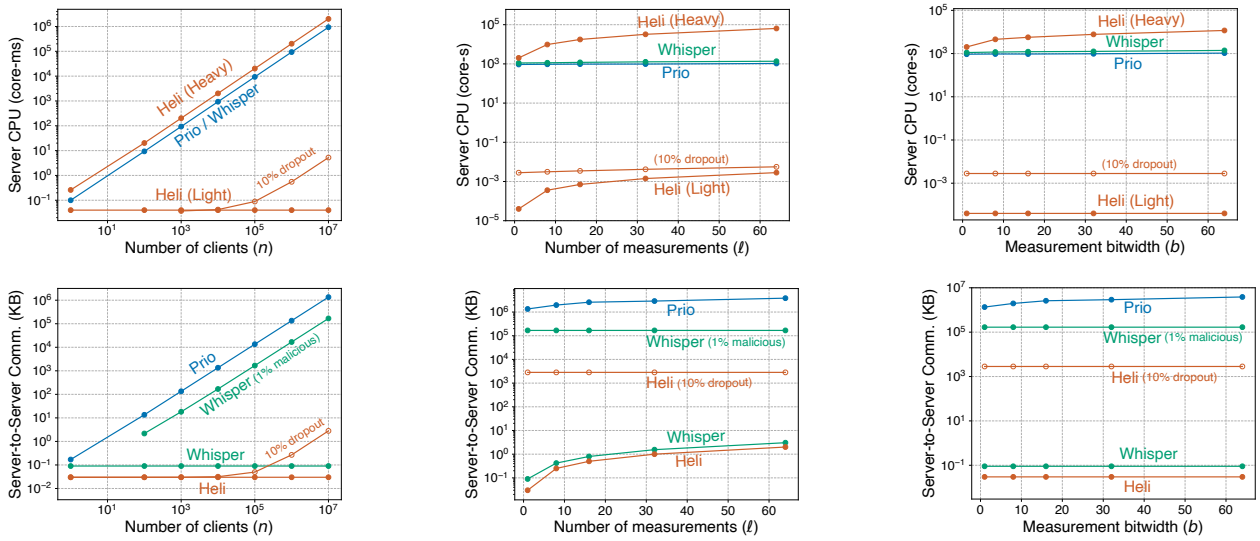

Figure 4: Heli's light server's work and total server-to-server communication scales with the number of dropped-out clients, not the total number of clients. The left plots show costs for aggregating $\ell = 1$ measurements of bitwidth $b = 1$ as the number of clients grows. The middle and right plots show costs for aggregating submissions from $n = 10,000,000$ clients as the measurement length or bitwidth grows. The line with hollow markers shows the light server's work when 10% of clients drop out. All plots use a logarithmic y-axis.

– Heli's computational cost, for the light server, and total server-to-server communication depends only on the number of offline clients in each round (Section 6.1).
– Heli's computational cost for the heavy server and clients grows linearly with the number of measurements, and is larger than those of Prio and Whisper (Section 6.2).
– When privately aggregating 32 1-bit integers across 10 million users, the AWS cost of operating a light server is five orders of magnitude cheaper than Prio and Whisper, but operating the heavy server is 38× more expensive (Section 6.3).

Implementation. We implemented Heli in 5,000 lines of Rust over the Curve25519 elliptic curve group [17]. Our code is open-source and available at `https://github.com/ryanleh/heli`.

Evaluation Setup. We compare against Prio [45, 49] and Whisper [94], two private aggregation schemes that achieve similar security goals to Heli. We use DivviUp's implementation of Prio [70] and Whisper's public implementation [102]. Whisper's server-to-server communication grows with the number of clients who submit invalid inputs (i.e., "malicious clients"). In Heli, "dropped-out clients" refers to both missing clients and clients who submit invalid inputs. We also compare Heli to a scheme that uses ElGamal encryption [62] in place of aggregation-only encryption to illustrate the overhead our approach introduces to protect against a malicious aggregator.

All aggregation schemes achieve 128-bits of computational security. We ran microbenchmarks on a single core of an AWS c7i.4xlarge instance (16 vCPUs, 32 GB of RAM). The heavy server's work is highly parallelizable, though our implementation did not exploit this. Our CPU benchmarks only account for cryptographic overheads, not network latency.

6.1 Costs of Heli's light server

Figure 4 shows that the light server's per-round computation and the server-to-server communication scales with the number of offline clients, not the total number of clients. In each aggregation round, the light server decrypts the aggregate ciphertext and performs a constant-time verification check. When d out of n clients drop out, the light server additionally receives $d \log n$ extra bits from the aggregator and computes the sum of d PRF (i.e., AES) evaluations.

Compared to a Prio or Whisper server, when aggregating a single bit over 10 million users (Figure 4, left), Heli's light server reduces the server CPU time by 23 million times when no clients drop out, and 178,000× when 10% of clients drop out. Heli servers communicate 44 million times fewer bits than Prio's servers when no clients drop out, and 480× fewer bits when 10% of clients drop out. When 1% of clients misbehave, Heli servers communicate 5.5 million times fewer bits than Whisper when no client drop out, and 60× fewer bits when 10% of clients drop out (Figure 4, bottom-left).

For a fixed number of users, the light server's work and communication between Heli servers increases linearly with the number of measurements (Figure 4, middle), but remains constant as the bitwidth grows (Figure 4, right).

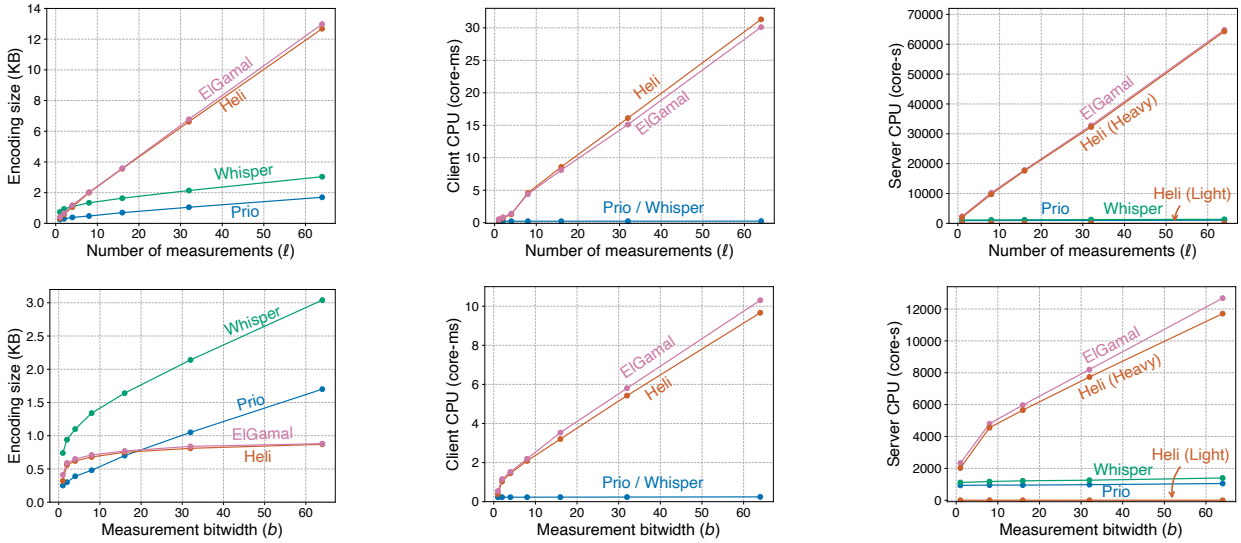

Figure 5: Compared to Prio and Whisper, Heli incurs higher costs on both the heavy server and clients as the number and bitwidth of measurements increase. Compared to the ElGamal-based scheme, Heli has a constant overhead in all metrics. The plots show encoding size, per-client CPU time, and server CPU time to aggregate $n = 10,000,000$ client submissions as the measurement type varies. The right-most plots are the same plots as the top-middle and top-right plots in Figure 4, but use a linear y-axis and omit the performance of Heli's light server.

Heli's light server remains efficient compared to a Prio or Whisper server, even when a large fraction of clients drop out. This is because computing missing clients' keys (one PRF evaluation per missing client) is several orders of magnitude faster than decrypting client submissions (λ group operations per online client). Moreover, if more than 50% of clients drop out, the heavy server can provide the list of *online* clients to the light server who directly computes the aggregated key. In Figure 6 we show that when aggregating a single bit over 10 million users, Heli's light server requires less server CPU time and server-to-server communication than Prio even when 50% of clients drop out.

6.2 Costs of Heli's heavy server and clients

Figure 5 shows the computational and communication costs of a Heli heavy server and client aggregating 10 million client submissions as the measurement type varies. The computational costs scale linearly with measurement length and bitwidth at a faster rate than Prio or Whisper, making large measurements more costly. Compared to the ElGamal baseline that only provides semi-honest security, Heli introduces at most a 5% overhead in client work, but reduces costs for all other metrics by up to 8%.

Compared to Prio and Whisper, the main source of additional cost for Heli's heavy server and clients comes from differences in the proof systems used to protect against malicious clients. Heli relies on discrete-log-based proofs (see Section 5.4) where the prover and verifier times scale linearly in the measurement length and bitwidth, and the encoding size scales linearly with the number of measurements (from the Schnorr proofs) and logarithmically with the measurement bitwidth (from Bulletproofs). In contrast, Prio and Whisper use information-theoretic proofs [26], with lower prover and verifier times, and whose encoding size scales sublinearly with the number and bitwidth of measurements. Table 1 summarizes the asymptotic costs of the different schemes.

The computational costs for Prio and Whisper's servers and clients is dominated by the cost of encrypting and decrypting client submissions: for the settings we consider, they take up 50–95% of the servers' and clients' total runtime, and so compute costs grow very slowly with the number and bitwidth of measurements (Figure 5, middle and right). Prio and Whisper's encoding sizes scale linearly with the number of measurements, but more slowly than Heli (Figure 5, top left). When the measurement bitwidth increases, the logarithmic scaling of Heli results in Prio and Whisper's encoding sizes eventually becoming larger (Figure 5, bottom left).

As the number of measurements grows, Heli's encoding becomes 0.4–7.5× larger than Prio and Whisper, the client encoding time grows by 2–138×, and the heavy server is 2–33× slower. As the measurement bitwidth grows, Heli's encoding is 0.3–1.4× larger than Prio and Whisper, the client encoding time is 4–43× slower, and the heavy server is 2–11× slower.

	Per-client costs				Per-server costs					
						Computation			*Communication*	
	Encrypt	Prove	PKE Wrap	Ciphertext Size		PKE Unwrap	Verify	Aggregate	Verify	Aggregate
Prio [45, 49]	$\ell\mathbb{F}$	$\sqrt{b\ell}\mathbb{F}$	$\lambda\mathbb{G}$	$(\ell+\sqrt{b\ell})\mathbb{F}$		$\lambda n\mathbb{G}$	$nb\ell\mathbb{F}$	$n\ell\mathbb{F}$	$n\sqrt{b\ell}\mathbb{F}$	$\ell\mathbb{F}$
Whisper [94]	$\ell\mathbb{F}$	$\sqrt{b\ell}\mathbb{F}$	$\lambda\mathbb{G}$	$(\ell+\sqrt{b\ell})\mathbb{F}$		$\lambda n\mathbb{G}$	$nb\ell\mathbb{F}$	$n\ell\mathbb{F}$	$m\mathbb{F}$	$\ell\mathbb{F}$
Heli (this work)	$\lambda\ell\mathbb{G}$	$\lambda b\ell\mathbb{G}$	$\lambda\mathbb{G}$	$(\ell+\log b)\mathbb{G}$	*Heavy:*	$\lambda n\mathbb{G}$	$\lambda nb\ell\mathbb{G}$	$(n\ell+\lambda\ell\sqrt{n2^b})\mathbb{G}$	0	$\ell\mathbb{G}$
					Light:	0	0	$\lambda\ell\mathbb{G}+(m+d)\mathsf{PRF}$	0	$\ell\mathbb{G}+(m+d)\log n$

Table 1: Asymptotic costs to aggregate ℓ measurements of b-bit integers over n clients, with d dropped-out clients, m malicious clients and security parameter λ. Heli's light server work and total server-to-server communication scales with the number of missing clients, not the total number of clients. Costs are expressed in terms of the number of field operations / elements ($\mathbb{F}$), group operations / elements ($\mathbb{G}$), and PRF evaluations (PRF). Clients use public-key encryption (PKE) over the same group as Heli to establish a secure channel with each server. In practice, group operations are 10× more expensive than field operations, and group elements are 2× the size of field elements. The ElGamal baseline has the same asymptotic costs as Heli.

6.3 Costs of an end-to-end deployment

We give end-to-end AWS deployment costs for privately aggregating vectors of 1-bit values across ten million clients when 10% drop out each aggregation round. We demonstrate that Heli reduces the AWS cost of running a decryptor by several orders of magnitude compared to Prio and Whisper, but increases the aggregator's cost. Boolean aggregates like these can track when application features trigger without revealing individual behavior. For example, Mozilla used them to measure how often specific websites triggered Firefox's tracker-blocking rules [21].

We deploy aggregation servers on AWS c7i.24xlarge instances (96 vCPUs, 192 GB RAM) and run Heli 's light server on a c7i.large instance (2 vCPUs, 4 GB RAM). In each deployment, the two servers are placed in us-east-1 and us-east-2. We generate and submit client reports from a separate c7i.24xlarge instance in us-east-2. The c7i.24xlarge instances provide 15 Gbit/s of network bandwidth; during aggregation, we fully saturate this bandwidth by pre-generating client reports and submitting them in batches. Based on current AWS pricing, the aggregating instances cost $4.28/hr, and outbound (egress) bandwidth is $0.90/GB.

Setup cost estimates. We estimate the setup costs for Heli's light server using Apple's App Attest [9] to verify clients. The main overheads of the verification are establishing a TLS connection, two ECDSA signature verifications for device attestation, a PRF evaluation to generate the client's key, and a group exponentiation to generate a commitment to the client's key. On our machine, the light server can process ten million clients in 2.5 hours of runtime and 1145 MB of network egress. Using current AWS estimates, this costs $0.32.

Aggregation cost estimates. In Table 2, we show the concrete costs of aggregating boolean vectors of length $\ell \in \{1, 32, 128\}$ when 10% of clients drop out. When aggregating vectors of length $\ell = 128$, Heli's heavy server costs $3.23, 13× more expensive than a single Prio server. Compared to Whisper, Heli's heavy server is 54× as expensive to run when no clients are malicious, and 49× more expensive when 1% of clients are malicious. Heli's light server costs 0.000057¢, which is 400,000× cheaper than Prio and at least 100,000× cheaper than Whisper.

For smaller vector sizes, Heli's relative performance to Prio and Whisper improves. For example, when aggregating a single boolean ($\ell = 1$), Heli's heavy server is 10× more expensive than a Prio server and at most 20× more expensive than Whisper, while the light server is 500,000× cheaper than Prio and at least 140,000× cheaper than Whisper.

Note that the total deployment cost and client costs of Heli are higher than Prio or Whisper. Moreover, because of the one-time setup, Heli 's light server may require multiple aggregation rounds before it becomes cheaper to run than a Prio or Whisper server. When aggregating boolean vectors of length $\ell = 32$ and verifying clients with Apple's App Attest [9], the light server becomes cheaper than a Prio server after two aggregation rounds, and cheaper than a Whisper server after six rounds. (Note that the setup cost is one-time per user; a single setup can support multiple applications by partitioning the round identifier space.)

7 Related work

Multi-server private aggregation. Many works build private aggregation schemes via a multiparty computation between multiple non-colluding servers [4, 11, 27, 45, 47, 49, 55, 61, 77, 93, 94]. These schemes require the servers' compute and communication to scale linearly with the number of clients. An alternative approach uses additively homomorphic encryption: one server aggregates client's encrypted submissions and a second server decrypts the aggregate [3, 34]. As discussed in Section 2.3, this approach is insecure in our setting where the aggregator sees the final result.

Number of Measurements		Per-client costs CPU (core-ms)	Upload (KB)		Server aggregation costs Wall Time (s)	Egress (KB)	AWS Cost (US cents)
$\ell = 1$	Prio	0.22	0.25		30	1350000	15
	Whisper	0.22	0.74		33	0.09	3.9
	Whisper (1% mal.)	0.22	0.74		33	168000	5.3
	Heli (this work)	0.35	0.39	*Heavy:*	669	2846	80
				Light:	0.0054	0.047	0.000014
$\ell = 32$	Prio	0.24	1.05		50	2910000	31
	Whisper	0.24	2.34		52	1.55	6.1
	Whisper (1% mal.)	0.24	2.34		52	168000	7.6
	Heli (this work)	8.67	6.64	*Heavy:*	952	2846	113
				Light:	0.0065	1	0.000025
$\ell = 128$	Prio	0.26	2.92		105	4160000	48
	Whisper	0.40	5.10		99	2.05	12
	Whisper (1% mal.)	0.40	5.10		99	168000	13
	Heli (this work)	34.4	24.8	*Heavy:*	2713	2846	323
				Light:	0.092	4	0.000057

Table 2: After a one-time setup, Heli's light server reduces the AWS cost of aggregation by up to five orders of magnitude, but increases the heavy server's cost by up to 54×. The table shows the CPU time, data egress, and current AWS cost in US cents to aggregate $\ell = 1, 32, 128$ measurements of bitwidth $b = 1$ over $n = 10,000,000$ clients, when 10% of clients are offline. Server costs for Prio and Whisper reflect the use of two servers, whereas Heli's costs are given separately for each server.

Single-server private aggregation. To avoid the need for multiple servers, a body of work builds private aggregation from a single, untrusted server by having clients do additional work [14, 15, 16, 24, 42, 72, 74, 80, 81, 83, 84, 85, 98]. The most efficient of these schemes [14, 16, 74, 80, 84, 85] draft a subset of clients into committees, concentrating most of the extra work on them. This design introduces a performance-security tradeoff: smaller committees reduce overall client costs, but require an adversary to compromise fewer clients to break security. In Heli, clients only need to send their input, and privacy is guaranteed for all honest clients, regardless of how many clients the adversary controls. However, this comes at the cost of requiring a second, non-colluding server.

LERNA [80] and OPA [74] use key-homomorphic PRFs in a similar manner to our aggregation-only encryption construction (Section 4.3). By collapsing their committee sizes to one (in the right way) and performing some additional preprocessing, you can extract an aggregation-only encryption scheme from their single-server constructions.

We highlight some key differences between Heli and three recent single-server aggregation schemes: OPA [74], Willow [16], and Armadillo [84]. Willow and OPA improve on prior work by requiring clients to send only a single message per aggregation task. However, in Willow, a malicious aggregator can learn the input of any honest client without detection by dropping client submissions and injecting fake ones to reach the decryption threshold. OPA protects against this type of attack, but is not robust: a single malicious client can cause the protocol to abort without detection. In principle, these attacks could be mitigated via zero-knowledge proofs or a more complicated MPC, but the authors do not describe or evaluate these extensions. Both protocols require a committee whose total work scales linearly with the number of clients. In Heli, clients also send only a single message, but their workload is independent of the number of clients. Moreover, Heli is secure against a malicious aggregator, and can identify and remove malicious clients without aborting. Armadillo similarly supports identifying and removing malicious clients without aborting, but assumes a semi-honest aggregator, requires three rounds of interaction between the server and clients, and requires *each* committee member to do work that scales linearly with the number of clients.

We view part of our contribution as recognizing that techniques from prior single-server aggregation works can apply to an asymmetric two-server setting. This setting combines the strengths of existing single- and two-server schemes: it offers a deployment model we expect to be more practical than traditional two-server designs, while maintaining strong privacy and robustness guarantees with low client overhead.

Differentially private aggregation. Some systems aim for differential privacy [54] instead of cryptographic privacy [8, 12, 44, 53, 56, 99, 104]. In these approaches, clients locally randomize their data before submission, which trades-off accuracy for privacy: more noise provides stronger privacy guarantees but the resulting measurement is less accurate. The additional noise also requires more complicated proof techniques to detect malicious clients [19]. In contrast, Heli reveals nothing beyond the final, exact aggregate. If output leakage is a concern, differential privacy can still be added in by the servers, as discussed in Section 5.6.

Aggregation from anonymity networks. PrivStats [92] gathers user statistics using an anonymity network build from onion routing [22, 50, 60]. Consequently, they are vulnerable to traffic-analysis attacks [75] while Heli protects against an adversary who compromises the entire network. Several

works [10, 20, 48, 91] defend against these attacks by using anonymity networks build from mixnets [40, 78] or DC-nets [41]. Prochlo [20] uses hardware enclaves to implement a mixnet, but these are vulnerable against enclave side-channel attacks [82]. Other approaches to building these types of networks require either each server to do quadratic work in the number of clients [41, 67] or run an expensive verifiable-shuffle routine [13, 78, 90, 103] that limit their practicality for large numbers of users.

Homomorphic MACs. Homomorphic message authentication codes (MACs) [35, 36, 39, 63, 76] and homomorphic signatures [5,28,29,37,38,69,71,101] allow an untrusted party to homomorphically evaluate a function on signed data and generate a succinct tag that certifies the function was applied correctly. Multi-key homomorphic MACs and homomorphic signatures [6, 57, 58, 59, 96] extend this notion to authenticate computation over data signed under different keys. Compared to aggregation-only encryption, homomorphic MACs and signatures do not require fixing a specific function during setup and provide a stronger security guarantee (an adversary cannot additively shift authenticated values). However, verification scales with the number of keys.

8 Conclusions

Heli demonstrates that a private-aggregation system can simultaneously (1) provide strong protection against a malicious server and (2) require only a single server to do per-statistic computational work scaling linearly with the number of clients. We close by mentioning a few intriguing directions for future work. Heli uses two servers; what is the cleanest way to extend Heli to support $k > 2$ servers (one heavy server and $k-1$ light ones) in a way that protects client privacy if any size-$(k-1)$ subset of the servers is malicious? Heli requires the light server to do work linear in the number of dropped-out clients in each aggregation round; is there a simple way to tolerate offline clients without requiring the light server to pay this cost?

Acknowledgments

We thank the USENIX Security reviewers for their detailed feedback, with special thanks to our anonymous shepherd for thoughtful guidance that substantially improved the final manuscript. We also thank Nickolai Zeldovich for early discussions of this work, as well as Christopher Patton and Tim Geoghegan for conversations that inspired the initial idea of Heli and for helpful feedback on an early draft. This work was supported in part by gifts from Amazon, Apple, Google, Meta, Microsoft, Mozilla, NSF Awards 2452708, 2140975, 2318701, 2141064 and a Sloan Research Fellowship. This work was done in part while H.C.G. and D.W. were visiting the Simons Institute for the Theory of Computing and supported in part by a grant from the UC Noyce Initiative.

Ethical considerations

In this work, we presented Heli, a new approach to privately gathering statistics over user data. By making it feasible for a single organization to operate a private-aggregation infrastructure supporting thousands of applications, Heli could significantly broaden the deployment of private aggregation. This raises several important ethical considerations, which we analyze through the lenses of "Respect for Persons" and "Beneficence," as defined in the Menlo Report [100].

First, Heli offers a path for companies to replace existing measurements that currently require intrusive access to sensitive user information. This would benefit users, whose privacy is better respected, as well as the companies themselves, who obtain the data they require without introducing additional risks of misuse or leakage. Since these measurements are already being performed, introducing private aggregation in this context constitutes a clear privacy improvement.

Second, Heli enables new types of measurements that were not possible previously due to legal or ethical concerns. For example, researchers could study the effects of personal devices without direct access to individual-level behavioral data, or companies could see if their application has disparate effects across different demographics groups. However, the ability to make new measurements does not, by itself, determine whether they are ethically appropriate: even with a guarantee of individual privacy, a statistic can still cause harm. Consequently, decisions about what measurements to pursue should involve consultation with data and privacy ethicists.

Finally, safeguards are needed to prevent misuse. Even if a given measurement is ethically acceptable, an organization might use Heli to justify increasingly intrusive data collection, and later remove Heli and measure the information directly. To reduce this risk, real-world deployments should be structured so that multiple parties have oversight and the ability to limit or shut down the system if necessary.

Overall, Heli has the potential to reduce existing privacy harms and enable new beneficial applications, but this potential can only be realized with appropriate ethical review and robust safeguards.

Open science

Our implementation of Heli is available at `https://github.com/ryanleh/heli` and `https://zenodo.org/records/17980904`.

References

[1] Internet security research group. https://www.abetterinternet.org/.

[2] Joshua Aas. Invited talk: Let's Encrypt: Ten years encrypting the web. Real World Cryptography Conference, slides at https://iacr.org/submit/files/slides/2025/rwc/rwc2025/inv1/inv1_slides.pdf, 2025.

[3] Ojaswi Acharya, Suvasree Biswas, Weiqi Feng, Adam O'Neill, and Arkady Yerukhimovich. Non-interactive verifiable aggregation. *Proc. Priv. Enhancing Technol.*, 2025(4):1055–1074, 2025.

[4] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In *SCN*, pages 516–539, 2022.

[5] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, Abhi Shelat, and Brent Waters. Computing on authenticated data. *J. Cryptol.*, 28(2):351–395, 2015.

[6] Gaspard Anthoine, David Balbás, and Dario Fiore. Fully-succinct multi-key homomorphic signatures from standard assumptions. In *CRYPTO*, pages 317–351, 2024.

[7] Apple. Learning iconic scenes with differential privacy. https://machinelearning.apple.com/research/scenes-differential-privacy.

[8] Apple. Learning with privacy at scale. "https://docs-assets.developer.apple.com/ml-research/papers/learning-with-privacy-at-scale.pdf".

[9] Apple. Validating apps that connect to your server. https://developer.apple.com/documentation/devicecheck/validating-apps-that-connect-to-your-server.

[10] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. Private summation in the multi-message shuffle model. In *ACM CCS*, pages 657–676, 2020.

[11] Laasya Bangalore, Mohammad Hossein Faghihi Sereshgi, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. Flag: A framework for lightweight robust secure aggregation. In *ASIA CCS*, pages 14–28, 2023.

[12] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, pages 127–135, 2015.

[13] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, pages 263–280, 2012.

[14] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. ACORN: input validation for secure aggregation. In *USENIX Security Symposium*, pages 4805–4822, 2023.

[15] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In *ACM CCS*, pages 1253–1269, 2020.

[16] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Phillipp Schoppmann. Willow: Secure aggregation with one-shot clients. In *CRYPTO*, pages 285–318, 2025.

[17] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *PKC*, pages 207–228, 2006.

[18] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT*, pages 321–340, 2013.

[19] Ari Biswas and Graham Cormode. Interactive proofs for differentially private counting. In *ACM CCS*, pages 1919–1933, 2023.

[20] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, pages 441–459, 2017.

[21] Mozilla Security Blog. Next steps in privacy-preserving telemetry with prio. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2019.

[22] Stevens Le Blond, David R. Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *ACM SIGCOMM*, pages 303–314, 2013.

[23] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112, 1988.

[24] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM CCS*, pages 1175–1191, 2017.

[25] Dan Boneh. The decision diffie-hellman problem. In *ANTS*, pages 48–63, 1998.

[26] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *CRYPTO*, pages 67–97, 2019.

[27] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE S&P*, pages 762–776, 2021.

[28] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, pages 149–168, 2011.

[29] Dan Boneh and David Mandell Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In *PKC*, pages 1–16, 2011.

[30] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *CRYPTO*, pages 410–428, 2013.

[31] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *EUROCRYPT*, pages 327–357, 2016.

[32] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, pages 315–334, 2018.

[33] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report 260, ETH Zurich, 1997.

[34] Claude Castelluccia, Aldar C.-F. Chan, Einar Mykletun, and Gene Tsudik. Efficient and provably secure aggregation of encrypted data in wireless sensor networks. *ACM Trans. Sens. Networks*, 5(3):20:1–20:36, 2009.

[35] Dario Catalano and Dario Fiore. Practical homomorphic MACs for arithmetic circuits. In *EUROCRYPT*, pages 336–352, 2013.

[36] Dario Catalano, Dario Fiore, Rosario Gennaro, and Luca Nizzardo. Generalizing homomorphic MACs for arithmetic circuits. In *PKC*, pages 538–555, 2014.

[37] Dario Catalano, Dario Fiore, and Ida Tucker. Additive-homomorphic functional commitments and applications to homomorphic signatures. In *ASIACRYPT*, pages 159–188, 2022.

[38] Dario Catalano, Dario Fiore, and Bogdan Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In *CRYPTO*, pages 371–389, 2014.

[39] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, and Jean-Pierre Hubaux. Verifiable encodings for secure homomorphic analytics. *CoRR*, abs/2207.14071, 2022.

[40] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24, 1981.

[41] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[42] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. In *ACM CCS*, pages 2535–2549, 2022.

[43] HeeWon Chung, Kyoohyung Han, Chanyang Ju, Myungsun Kim, and Jae Hong Seo. Bulletproofs+: Shorter proofs for a privacy-enhanced distributed ledger. *IEEE Access*, 10:42067–42082, 2022.

[44] Graham Cormode and Akash Bharadwaj. Sample-and-threshold differential privacy: Histograms and applications. In *AISTATS*, pages 1420–1431, 2022.

[45] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[46] Henry Corrigan-Gibbs and Dmitry Kogan. The discrete-logarithm problem with preprocessing. In *EUROCRYPT*, pages 415–447, 2018.

[47] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *ACM Workshop on Smart Energy Grid Security*, pages 75–80, 2013.

[48] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. *Proc. Priv. Enhancing Technol.*, 2017(3):21, 2017.

[49] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Proc. Priv. Enhancing Technol.*, 2023(4):578–592, 2023.

[50] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320, 2004.

[51] Divvi Up. https://divviup.org/.

[52] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *CRYPTO*, pages 93–122, 2016.

[53] Yitao Duan, NetEase Youdao, John F. Canny, and Justin Z. Zhan. P4P: practical large-scale privacy-preserving distributed computation robust against malicious users. In *USENIX Security Symposium*, pages 207–222, 2010.

[54] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, 2006.

[55] Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private collection of traffic statistics for anonymous communication networks. In *ACM CCS*, pages 1068–1079, 2014.

[56] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *ACM CCS*, pages 1054–1067, 2014.

[57] Shuai Feng, Shuaijianni Xu, and Liang Feng Zhang. Multi-key homomorphic MACs with efficient verification for quadratic arithmetic circuits. In *ASIA CCS*, pages 17–27, 2022.

[58] Dario Fiore, Aikaterini Mitrokotsa, Luca Nizzardo, and Elena Pagnin. Multi-key homomorphic authenticators. In *ASIACRYPT*, pages 499–530, 2016.

[59] Dario Fiore and Elena Pagnin. Matrioska: A compiler for multi-key homomorphic signatures. In *SCN*, pages 43–62, 2018.

[60] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In *ACM CCS*, pages 193–206, 2002.

[61] David Froelicher, Juan Ramón Troncoso-Pastoriza, Joao Sa Sousa, and Jean-Pierre Hubaux. Drynx: Decentralized, secure, verifiable system for statistical queries and machine learning on distributed datasets. *IEEE Trans. Inf. Forensics Secur.*, 15:3035–3050, 2020.

[62] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984.

[63] Rosario Gennaro and Daniel Wichs. Fully homomorphic message authenticators. In *ASIACRYPT*, pages 301–320, 2013.

[64] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[65] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[66] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304, 1985.

[67] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *EUROCRYPT*, pages 456–473, 2004.

[68] Google. Exposure notifications: Help slow the spread of COVID-19, with one step on your phone. "https://www.google.com/covid19/exposurenotifications".

[69] Sergey Gorbunov, Vinod Vaikuntanathan, and Daniel Wichs. Leveled fully homomorphic signatures from standard lattices. In *STOC*, pages 469–477, 2015.

[70] https://github.com/divviup/janus.

[71] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David A. Wagner. Homomorphic signature schemes. In *CT-RSA*, pages 244–262, 2002.

[72] Swanand Kadhe, Nived Rajaraman, Onur Ozan Koyluoglu, and Kannan Ramchandran. FastSecAgg: Scalable secure aggregation for privacy-preserving federated learning. *CoRR*, abs/2009.11248, 2020.

[73] Alan F. Karr, Xiaodong Lin, Ashish P. Sanil, and Jerome P. Reiter. Regression on distributed databases via secure multi-party computation. In *Proceedings of the 2004 Annual National Conference on Digital Government Research*, 2004.

[74] Harish Karthikeyan and Antigoni Polychroniadou. OPA: one-shot private aggregation with single client interaction and its applications to federated learning. *CoRR*, abs/2410.22303, 2024.

[75] Ishan Karunanayake, Nadeem Ahmed, Robert A. Malaney, Rafiqul Islam, and Sanjay K. Jha. De-anonymisation attacks on tor: A survey. *IEEE Commun. Surv. Tutorials*, 23(4):2324–2350, 2021.

[76] Shuichi Katsumata, Ryo Nishimaki, Shota Yamada, and Takashi Yamakawa. Designated verifier/prover and preprocessing NIZKs from Diffie-Hellman assumptions. In *EUROCRYPT*, pages 622–651, 2019.

[77] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *PETS*, pages 175–191, 2011.

[78] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proc. Priv. Enhancing Technol.*, 2016(2):115–134, 2016.

[79] Ryan Lehmkuhl, Henry Corrigan-Gibbs, Emma Dauterman, and David J. Wu. Heli: Heavy-light private aggregation. Cryptology ePrint Archive, Paper 2026/59.

[80] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. LERNA: secure single-server aggregation via key-homomorphic masking. In *ASIACRYPT*, pages 302–334, 2023.

[81] Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. SASH: Efficient Secure Aggregation Based on SHPRG for Federated Learning. 2021/2022.

[82] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2022.

[83] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. RoFL: Robustness of secure federated learning. In *IEEE S&P*, pages 453–476, 2023.

[84] Yiping Ma, Yue Guo, Harish Karthikeyan, and Antigoni Polychroniadou. Armadillo: Robust single-server secure aggregation for federated learning with input validation. In *ACM CCS*, pages 2219–2233, 2025.

[85] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. In *IEEE S&P*, pages 477–496, 2023.

[86] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *NDSS*, 2016.

[87] Joseph P. Mihalcik. An analysis of algorithms for solving discrete logarithms in fixed groups. Master's thesis, Naval Postgraduate School, 2010.

[88] Mozilla. Built for privacy: Partnering to deploy oblivious HTTP and Prio in Firefox. `https://blog.mozilla.org/en/products/firefox/partnership-ohttp-prio/`.

[89] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In *EUROCRYPT*, pages 327–346, 1999.

[90] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM CCS*, pages 116–125, 2001.

[91] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In *ACM CCS*, pages 1570–1581, 2015.

[92] Raluca A. Popa, Andrew J. Blumberg, Hari Balakrishnan, and Frank H. Li. Privacy and accountability for location-based aggregate statistics. In *ACM CCS*, pages 653–666, 2011.

[93] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. ELSA: secure aggregation for federated learning with malicious actors. In *IEEE S&P*, pages 1961–1979, 2023.

[94] Mayank Rathee, Yuwen Zhang, Henry Corrigan-Gibbs, and Raluca Ada Popa. Private analytics via streaming, sketching, and silently verifiable proofs. In *IEEE S&P*, pages 3072–3090, 2024.

[95] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11), 1978.

[96] Lucas Schabhüser, Denis Butin, and Johannes Buchmann. Context hiding multi-key linearly homomorphic authenticators. In *CT-RSA*, pages 493–513, 2019.

[97] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 1989.

[98] Jinhyun So, Corey J. Nolet, Chien-Sheng Yang, Songze Li, Qian Yu, Ramy E. Ali, Basak Guler, and Salman Avestimehr. LightSecAgg: a lightweight and versatile design for secure aggregation in federated learning. In *MLSys*, 2022.

[99] Timothy Stevens, Christian Skalka, Christelle Vincent, John H. Ring, Samuel Clark, and Joseph P. Near. Efficient differentially private secure aggregation for federated learning via hardness of learning with errors. In *USENIX Security Symposium*, pages 1379–1395, 2022.

[100] Science U.S. Department of Homeland Security and Cyber Security Division Technology Directorate. The menlo report: Ethical principles guiding information and communication technology research. Technical report, August 2012.

[101] Hoeteck Wee and David J. Wu. Succinct functional commitments for circuits from k-Lin. In *EUROCRYPT*, pages 280–310, 2024.

[102] `https://github.com/ucbsky/whisper`.

[103] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, pages 179–182, 2012.

[104] Mingxun Zhou, Tianhao Wang, T.-H. Hubert Chan, Giulia Fanti, and Elaine Shi. Locally differentially private sparse vector aggregation. In *IEEE S&P*, pages 422–439, 2022.

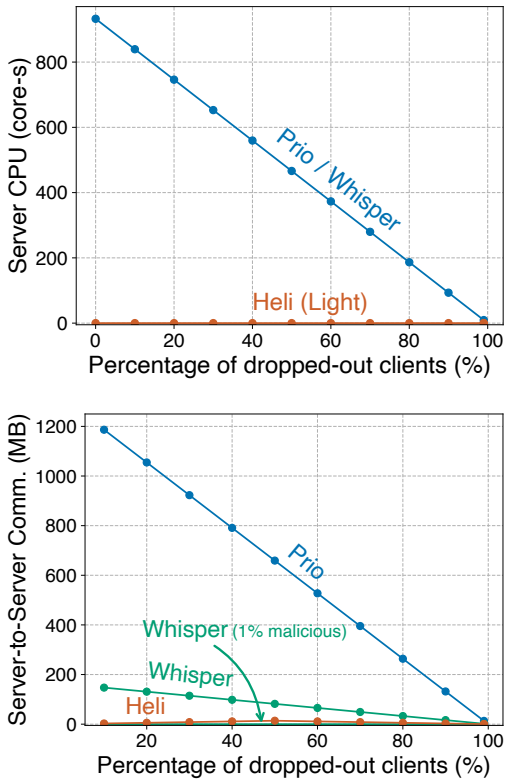

Figure 6: Heli's light server requires less total work and communication than a Prio server, even in its worst-case scenario where 50% of users drop out. The plots shows server CPU time and server-to-server communication to aggregate $\ell = 1$ measurements of bitwidth $b = 1$ over $n = 10,000,000$ clients as the percentage of dropped-out clients increases.

Additional notation

The set $\mathbb{Z}^{\geq 0} = \{1, 2, 3, \ldots, \}$ denotes the natural numbers. We use boldfaced lowercase variable names (e.g., $\mathbf{v}$) to denote vectors and boldfaced uppercase names (e.g., $\mathbf{M}$) to denote matrices. For a matrix $\mathbf{M}$, we use $\mathbf{M}_{i,j}$ to denote the element in the i^{th} row and j^{th} column of the matrix. We write $\mathcal{D}_0 \overset{c}{\approx} \mathcal{D}_1$ to denote that the distributions $\mathcal{D}_0$ and $\mathcal{D}_1$ are computationally indistinguishable. For vectors $\mathbf{u}$ and $\mathbf{v}$, we denote their concatenation as $\mathbf{u}\|\mathbf{v}$. We write $\text{negl}(\lambda)$ to denote a function that is bounded by a negligible function in λ (a function that is $o(\lambda^{-c})$ for all $c \in \mathbb{Z}^{\geq 0}$).

A Details on aggregation-only encryption

In this section, we formally show that Construction 4.1 satisfies security against a malicious aggregator and a malicious decryptor when instantiated with any secure key-homomorphic PRF (e.g., the Naor-Pinkas-Reingold scheme in Construction 3.1).

A.1 Definitions

We first formally define correctness and security notions for aggregation-only encryption.

Definition A.1 (Aggregation-only encryption: Correctness)**.** Formally, let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Apply}, \mathsf{Dec})$ be an aggregation-only encryption scheme for message space $\mathcal{M}$ (with default value m^*), round-identifier space $\mathcal{R}$, and function class $\mathcal{F}$. We say that $\mathcal{E}$ is *correct* if for all functions $f \in \mathcal{F}$, all drop-out sets $D \subseteq [n]$, all messages $m_i \in \mathcal{M}$ for $i \in [n] \setminus D$, and all round identifiers $r \in \mathcal{R}$, the following probability is at least $1 - \text{negl}(\lambda)$:

$$\Pr\left[m = m' : \begin{array}{r} (\mathsf{sk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow \mathsf{Gen}(1^\lambda, f) \\ \forall i \in [n] \setminus D : \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{sk}_i, r, m_i) \\ \forall i \in D : m_i \leftarrow m^* \\ \mathsf{ct} \leftarrow \mathsf{Apply}(f, D, \{\mathsf{ct}_i\}_{i \in [n] \setminus D}) \\ m \leftarrow \mathsf{Dec}(\mathsf{sk}, r, D, \mathsf{ct}) \\ m' \leftarrow f(m_1, \ldots, m_n) \end{array} \right].$$

We use the following definition of security, which references the security experiments in Figure 7.

Definition A.2 (Aggregation-only encryption: Security)**.** Let $\mathcal{E}$ be an aggregation-only encryption scheme for message space $\mathcal{M}$, and function class $\mathcal{F}$. We say that $\mathcal{E}$ is *secure against a malicious decryptor* if for every efficient adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that for all time bounds $T \in \mathbb{Z}^{\geq 0}$ and for all messages $\mathbf{M} \in \mathcal{M}^{T \times n}$, the distribution families defined in Figure 7 (parameterized by λ) are computationally indistinguishable:

$$\text{REAL}^{\mathsf{Dec}}_{\mathcal{E},\mathcal{A},T,\mathbf{M}} \overset{c}{\approx} \text{IDEAL}^{\mathsf{Dec}}_{\mathcal{E},\mathcal{S},T,\mathbf{M}}.$$

We further say that $\mathcal{E}$ is *secure against a malicious aggregator* if, under the same conditions, we have

$$\text{REAL}^{\mathsf{Agg}}_{\mathcal{E},\mathcal{A},T,\mathbf{M}} \overset{c}{\approx} \text{IDEAL}^{\mathsf{Agg}}_{\mathcal{E},\mathcal{S},T,\mathbf{M}},$$

which are again defined in Figure 7.

All experiments are parameterized by:

- an aggregation-only encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Apply}, \mathsf{Dec})$ for message space $\mathcal{M}$, round-identifier space $\mathcal{R}$, and function class $\mathcal{F} \subseteq \mathsf{Funs}[\mathcal{M}^*, \mathcal{M}]$ (with default value m^* for dropped-out clients),
- an algorithm, which is the adversary $\mathcal{A}$ in the real world, and is the simulator $\mathcal{S}$ in the ideal world,
- a number of rounds $T \in \mathbb{Z}^{\geq 0}$, and
- a matrix of messages $\mathbf{M} \in \mathcal{M}^{T \times n}$ representing honest parties' inputs.

All experiments take a security parameter $\lambda \in \mathbb{Z}^{\geq 0}$ as input and the final output is a single bit $b \in \{0, 1\}$.

Security experiments: Malicious aggregator.

$\underline{\text{REAL}^{\mathsf{Agg}}_{\mathcal{E},\mathcal{A},T,\mathbf{M}}(\lambda):}$

// Adversary chooses function $f \in \mathcal{F}$ and
// compromised set $\mathcal{C} \subseteq [n]$.

- $(\mathsf{st}, f, \mathcal{C}) \leftarrow \mathcal{A}(1^\lambda)$.
- $(\mathsf{sk}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n) \leftarrow \mathsf{Gen}(1^\lambda, f)$.

// Adversary picks round identifiers $r_1, \ldots, r_T \in \mathcal{R}$.

- $(\mathsf{st}, r_1, \ldots, r_T) \leftarrow \mathcal{A}(\mathsf{st}, \{\mathsf{ek}_i\}_{i \in \mathcal{C}})$.
- If $\{r_1, \ldots, r_T\}$ are not distinct: Output $\perp$.

// Adversary gets honest ciphertexts.

- $(\mathsf{st}, D, \mathsf{ct}) \leftarrow \mathcal{A}(\mathsf{st}, \{\mathsf{Enc}(\mathsf{ek}_i, r_t, \mathbf{M}_{t,i}\}_{t \in [T], i \in [n] \setminus \mathcal{C}})$.
- For $t \in [T]$:
  - $m' \leftarrow \mathsf{Dec}(\mathsf{sk}, r_t, D, \mathsf{ct})$.
  - $(\mathsf{st}, D, \mathsf{ct}) \leftarrow \mathcal{A}(\mathsf{st}, m')$.
- Output st.

$\underline{\text{IDEAL}^{\mathsf{Agg}}_{\mathcal{E},\mathcal{S},T,\mathbf{M}}(\lambda):}$

- $(\mathsf{st}, f, \mathcal{C}) \leftarrow \mathcal{S}(1^\lambda)$.
- For $t \in [T]$:
  - $(\mathsf{st}, D) \leftarrow \mathcal{S}(\mathsf{st})$.
  - For $i \in [n]$:
    - If $i \in D$: $m_i \leftarrow m^*$.
    - If $i \in \mathcal{C} \setminus D$: $(\mathsf{st}, m_i) \leftarrow \mathcal{S}(\mathsf{st})$.
    - Otherwise: $m_i \leftarrow \mathbf{M}_{t,i}$.
  - $m \leftarrow f(m_1, \ldots, m_n)$.
  - $\mathsf{st} \leftarrow \mathcal{S}(\mathsf{st}, m)$.
- Output st.

Security experiments: Malicious decryptor.

$\underline{\text{REAL}^{\mathsf{Dec}}_{\mathcal{E},\mathcal{A},T,\mathbf{M}}(\lambda):}$

// Adversary chooses a function $f \in \mathcal{F}$,
// compromised set $\mathcal{C} \subseteq [n]$, keys ek_i
// and initial round identifier $r_1 \in \mathcal{R}$.

- $(\mathsf{st}, f, \mathcal{C}, \mathsf{sk}, \mathsf{ek}_1, \ldots, \mathsf{ek}_n, D, r_1) \leftarrow \mathcal{A}(1^\lambda)$.
- For $t \in [T]$:
  - For $i \in [n]$:
    - If $i \in \mathcal{C}$: $(\mathsf{st}, \mathsf{ct}_i) \leftarrow \mathcal{A}(\mathsf{st})$.
    - Otherwise: $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{ek}_i, r_t, \mathbf{M}_{t,i})$.
  - $\mathsf{ct} \leftarrow \mathsf{Apply}(f, D, \{\mathsf{ct}_i\}_{i \in [n] \setminus D})$.
  - $(\mathsf{st}, D, r_{t+1}) \leftarrow \mathcal{A}(\mathsf{st}, \mathsf{ct})$.
  - If $\{r_1, \ldots, r_{t+1}\}$ are not distinct: Output $\perp$.
- Output st.

$\underline{\text{IDEAL}^{\mathsf{Dec}}_{\mathcal{E},\mathcal{S},T,\mathbf{M}}(\lambda):}$

// Exactly the same as above ideal case.

- Output $\text{IDEAL}^{\mathsf{Agg}}_{\mathcal{E},\mathcal{S},T,\mathbf{M}}(\lambda)$.

Figure 7: Security experiments for defining security against a malicious aggregator and decryptor.