Private Analytics via Streaming, Sketching, and Silently Verifiable Proofs

Yuwen Zhang, with external collaborators Mayank Rathee, Raluca Ada Popa, and Henry Corrigan Gibbs

***Abstract*—We present Whisper, a system for privacy-preserving collection of aggregate statistics. Like prior systems, a Whisper deployment consists of a small set of non-colluding servers; these servers compute aggregate statistics over data from a large number of users without learning the data of any individual user. Whisper's main contribution is that its server-to-server communication cost and its server-side storage costs scale *sublinearly* with the total number of users. In particular, prior systems required the servers to exchange a few bits of information to verify the well-formedness of each client submission. In contrast, Whisper uses *silently verifiable proofs*, a new type of proof system on secret-shared data that allows the servers to verify an arbitrarily large batch of proofs by exchanging a single 128-bit string. This improvement comes with increased client-to-server communication, which, in cloud computing, is typically cheaper (or even free) than the cost of egress for server-to-server communication. To reduce server storage, Whisper approximates certain statistics using small-space sketching data structures. Applying randomized sketches in an environment with adversarial clients requires a careful and novel security analysis. In a deployment with two servers and 100,000 clients of which 1% are malicious, Whisper can improve server-to-server communication for vector sum by three orders of magnitude while each client's communication increases by only 10%.**

1. Introduction

Private-aggregation systems make it possible to compute aggregate statistics about a population of devices, while revealing no information—beyond the aggregate statistic itself—about any device's data. These systems make it possible to privately collect information on user behavior [3], public health trends [7], [52], and device telemetry [69] at million-user scale.

In this paper, we focus on private-aggregation systems based on multi-party computation techniques [2], [9], [17], [32], [34], [36], [38], [44], [48], [49], [56], [62], [67], [68]. These systems require a small set of infrastructure providers ("servers"); the systems protect client privacy as long as an adversary cannot compromise all servers. Deployments of private aggregation at Apple [3], Google [7], Mozilla [69], and others [38] use this approach.

In a run of one such private-aggregation protocol, each user splits its data using a cryptographic secret-sharing scheme, and sends one share to each server. In addition, each user sends the servers a zero-knowledge proof attesting that its secret shares are well-formed. After receiving the data submissions and validity proofs from a large number of clients, the servers verify each proof, and then aggregate the valid submissions to compute the statistic of interest.

An annoyance in prior systems [2], [9], [32], [36], [38], [56] is that the servers must exchange messages to check *each client's* validity proof, so the server-to-server communication cost is linear in the number of clients. When the number of clients is in the millions, this server-to-server communication cost is significant.

More recent systems [17], [58], [68] support computing the "heavy-hitter" statistic: each client holds a string and the statistic computes the set of most popular client-held strings. This statistic is useful when the universe of possible client submission is large—for example, when computing the set of URLs that most often cause a user's browser to crash.

When computing heavy hitters, existing multiparty-computation-based systems [17], [58], [68] require the servers to store an amount of secret state that grows linearly with the number of clients. When the client submissions arrive in a stream [1], the servers cannot begin processing the first client submission until the last submission arrives. As a result, as the number of client uploads increases, the servers' memory and storage requirements balloon.

We present Whisper, a system for the privacy-preserving collection of aggregate statistics that has server-to-server communication and server storage costs that are *sublinear* in the number of users. Whisper provides these efficiency properties while supporting the computation both of simple arithmetic statistics and of heavy hitters. Whisper operates in the same deployment model as existing systems [32], [36], [58] (Figure 1) and provides the same privacy property: if there is at least one honest server, no adversarial coalition of malicious clients and servers can learn any information about honest clients' data, beyond what the aggregate statistics themselves leak.

Silently verifiable proofs. To reduce server-to-server communication in Whisper, we introduce *silently* verifiable proofs, a new type of zero-knowledge proof system on secret-shared data [16]. In a silently verifiable proof system, the verifiers can verify a batch of proofs by exchanging a single field element. This batch verification is possible even when the provers are mutually distrusting and when each prover is proving a different statement. Clients in Whisper use silently verifiable proofs to convince the servers that their data submissions are well formed; the servers can check arbitrarily large batches of proofs using only a few bits of server-to-server communication.

Most of the work to develop the cryptography behind

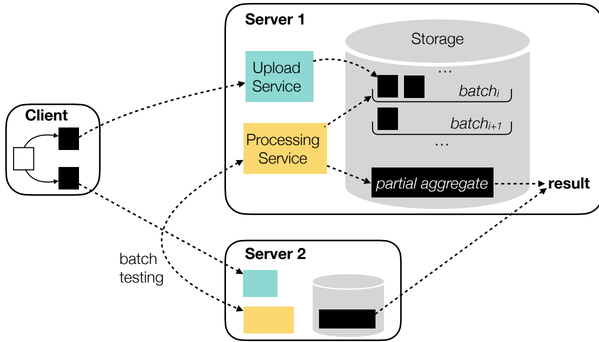

Figure 1: Whisper's server architecture. Clients split their data using a secret-sharing scheme and send one share to each server. The servers process submissions in batches.

Silently Verifiable Proofs was finished before this class. We will provide a brief overview of their syntax and and construction, but we will mostly focus on the systems-level consequences of their use.

Privately streaming heavy hitters. To avoid needing to store per-client state in our private heavy-hitters computation, we use a small-space sketching data structure [72]. In doing so, we give up on computing the exact heavy hitters, and instead settle for a good approximation—we expect this trade-off to be acceptable in many applications. Outside of this class, we formally bound the effect that a malicious client can have on the final computation of the heavy hitters.

We implement Whisper on top of ISRG's `libprio-rs` library [39]. In our evaluation where two servers aggregate 1024-sized vectors across 100,000 clients of which 1% are malicious, each server in Whisper only sends 0.2 MB compared to 415 MB for state-of-the-art Prio3 [39]. In achieving this, our per-client communication increases to 303 KB from 274 KB for Prio3. This trade-off is most appealing in cloud deployments, where ingress is free and egress is costly. We estimate up to 3× reduction in server operating costs for certain statistics. We additionally evaluate our system's performance over the Android privacy preserving exposure notification, a collection of 14 prio-style aggregate statistics originally When the same servers compute heavy hitters over a stream of 1.75 million client uploads, our baseline Poplar [17] overruns the 64 GB memory at the servers and takes four days to finish, while Whisper takes about two hours and recovers all the heavy hitters with probability at least 0.999, while taking only 15 secs to finish after receiving the last upload. Streaming the heavy hitter computation in Whisper comes at a 14-17× increase in client communication over Poplar, however, it stays under 500 KB.

2. System Overview

In this section, we outline Whisper's system architecture, capabilities, and security properties.

2.1. System Model

A Whisper deployment consists of two or more logical servers, and a large number of clients. All the communication happens over TLS-protected network channels.

Servers. Each logical server in Whisper server runs in its own administrative domain, separate from all other servers in the system. A logical Whisper server can consist of a large number of physical servers or cloud instances. For conciseness, we use "server" to refer to a logical server. We assume that all participants in the system have the cryptographic public keys of each server in the system. The servers jointly compute the same set of aggregate statistics on the users' data. There are v servers (*v*erifiers).

Clients. Each client holds a piece of private data; the servers compute aggregate statistics over all clients' data. Clients in Whisper communicate with each of the Whisper servers and do not communicate with other clients. We assume that the clients have a means to authenticate to the servers [5].

2.2. Architecture

Whisper computes aggregate statistics in a sequence of time *epochs*: at the end of each epoch, the servers publish a set of aggregate statistics computed over the data of the clients participating during the just-completed epoch. The protocol flow in each epoch works in the following three steps, depicted in Figure 1.

Step 1: Client data submission. Each client authenticates to the *Upload Service* at each server, which associates this client with an id. The client uploads an encoding of its private data with a silently verifiable proof of valid encoding by sending a single message to each server.

The Upload Service associates each message with a specific batch of submissions, a batch corresponding to a time interval. We need to ensure that each client uploads to the *same* batch on each server. A malicious client can try to upload in v different batches at the v servers to cause v times more work for the servers. At the same time, we do not want the "silent" servers to communicate per client to reach consensus. To prevent this issue, Whisper has each client first submit its upload to the first server. This server will verify that this client id has not uploaded already in the epoch. It will assign this message to a batch and will return a signed acknowledgment σ_{ack} that covers the batch identifier and the client id. The client will then upload to the other servers to the same batch by presenting σ_{ack}. This guarantees that every client input belongs to the same

Step 2: Server data validation and aggregation. After receiving client submissions, the servers check that they are well formed using the silently verifiable proof in each submission. To keep the server state from growing, Whisper servers verify client submissions in batches as they arrive within the epoch. The *Processing Service* processes each batch. It first tests the validity of the submissions in the batch by running the batch-verification routine of the silently verifiable proofs. In the optimistic case—when all clients in

a batch are honest—the entire validity check requires the servers to exchange a single short (128-bit) field element. If any proof in the batch is invalid, the servers will identify the failing proof via group-testing techniques [41]; they will discard the corresponding malformed submissions. These steps require interacting with the corresponding Processing Service on the other servers. It then aggregates the values in the batch into a running partial aggregate.

Step 3: Publishing the aggregate statistic. After the servers process all input batches, the Processing Service combines the resulting aggregate with the aggregates on the other servers to obtain the aggregate statistic.

2.3. Supported statistics

The configuration of a Whisper deployment specifies which aggregate statistic f the system will compute in each epoch. Following prior private-aggregation systems [17], [32], [36], Whisper supports any aggregate statistic that can be computed via a verifiable *additive encoding* of the client's data [32], [54]. We discuss additive encodings in more detail in §4. Using encoding techniques from prior work [22], [30], [32], [44], [67], [73], Whisper supports the following aggregation functions:

- *Basic statistics:* SUM, MEAN, VARIANCE, STDDEV, MIN/MAX (over small domains)
- *Counting:* FREQUENCY COUNT, APPROXIMATE FREQUENCY
- *Boolean operations:* AND, OR
- *Machine learning:* LINEAR REGRESSION, r^2 COEFFICIENT

As one of our technical contributions, we show that Whisper can also support computation of approximate *heavy hitters* (popular strings) §5.

In many cases, Whisper reveals to the servers slightly more information about the client inputs $(x_1, \dots, x_n)$ than the aggregate statistic $f(x_1, \dots, x_n)$ itself. For example, for SUM, there is no extra leakage, while private-aggregation schemes for VARIANCE additionally leak the mean for efficiency [32]. In this case, as in prior work, we define the *leakage* $\hat{f}(x_1, \dots, x_n)$ of the encoding to capture this extra information. In all cases in Whisper, the leakage function is *symmetric* in its inputs—so the leakage reveals no information about which client i held which private input x_i.

2.4. Security properties

Whisper's security properties are similar to existing privacy-preserving systems for collecting aggregate statistics. We describe these properties in more detail in Sections 4.2 and 5; we sketch them here. All of the security properties are relative to an aggregate statistic f and an associated leakage function $\hat{f}$.

- *Privacy*. As long as one server is honest, no server or malicious client learns any information about the honest clients' data $x_1, \dots, x_n$, except what can be inferred from the aggregate statistic $f(x_1, \dots, x_n)$ and the leakage function $\hat{f}(x_1, \dots, x_n)$. All the statistics f that Whisper supports and their leakage functions $\hat{f}$ are symmetric in their inputs, and therefore, the output reveals no information about which client submitted which input.
- *Correctness against malicious clients*. If all the servers are honest, then a small set of malicious clients can only affect the aggregate statistic f by misreporting their private data. When f computes heavy hitters, we allow malicious clients to introduce some small additional error in the output with low probability.

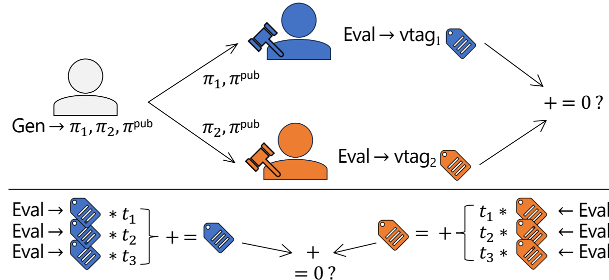

Figure 2: Silently verifiable proofs with batch verification.

For privacy, it is important that "enough" honest clients participate in each epoch. This ensures that $f(x_1, \dots, x_n)$ and $\hat{f}(x_1, \dots, x_n)$ don't reveal any private information about honest clients' inputs. For example, if there is a single honest client in an epoch, the output can trivially leak the client's data. Noising the aggregate statistic to provide differential privacy [43], [67] gives some protection in this case. To limit the number of malicious clients, as in prior works [16], [17], [32], [67], we assume that the servers employ Sybil-protection mechanisms [5], [6], [38], [75].

We assume that pairwise authenticated and encrypted channels exist from clients to servers and between the servers. We make no synchrony requirement and the adversary can observe all network links.

3. Silently Verifiable Proofs on Secret Shares

A silently verifiable proof system is a new type of zero-knowledge proof system on secret-shared data that allows a set of verifiers to check an arbitrarily large batch of proofs, from independent provers, with verifier-to-verifier communication cost *constant* in the batch size.

We first recall the definition of a zero-knowledge proof on secret-shared data [16]. Such a proof system is a protocol that takes place between:

- a *prover*, holding an input $x \in \mathbb{F}^n$, for a finite field $\mathbb{F}$ and input size n,
- many *verifiers*, where each verifier holds an additive secret share of the input x.

The protocol allows the prover to convince the verifiers that the input x satisfies a public predicate—i.e., that the input x is in some language $\mathcal{L} \subseteq \mathbb{F}^n$—while revealing nothing about the input x apart from the fact that $x \in \mathcal{L}$.

We consider a flavor of zero-knowledge proofs on secret-shared data that has a very simple communication pattern:

1) the prover sends each verifier a single message,
2) the verifiers each broadcast a single message to the other verifiers, and
3) each verifier runs some computation on these received messages to determine whether to accept or reject the proof.

Many existing proof systems have this structure [16], [32], [36]. In practice, a designated verifier receives the messages from all verifiers and decides to accept or reject the proof.

A *silently verifiable proof system* is a special zero-knowledge proof on secret-shared data in which the verifiers' decision of whether to accept or reject the proof is a *linear* function of the broadcasted messages. As we discuss in §3.2, silently verifiable proofs allow verifiers to check a large batch of proofs at once, with minimal verifier-to-verifier communication.

3.1. Definition

We provide an informal definition of silently verifiable proof systems in the information-theoretic setting. That is, we require the proof systems to be secure against computationally unbounded prover and verifiers. Later on, we will consider computationally-secure variants of these proof systems—in that setting, we consider infinite families of languages $\mathcal{L} = \{\mathcal{L}_\lambda\}_{\lambda=1}^{\infty}$, we require all algorithms to run in time $\mathsf{poly}(\lambda)$, and we prove security against adversaries that run in time $\mathsf{poly}(\lambda)$.

Our definition of zero-knowledge proofs on secret-shared data closely follows those in prior work [16], [32], [36]. The key differences are:

1) our proofs have a "public part" that the prover sends to all verifiers, in addition to a per-verifier "secret part," and
2) we only consider non-interactive proof systems—in which the prover sends a single message to each verifier.

Notation. Throughout, for a natural number n, we use $[n]$ to denote the set $\{1, \dots, n\}$.

Syntax: Zero-knowledge proof on secret-shared data. For a finite field $\mathbb{F}$, input size n, tag size q, and language $\mathcal{L} \subset \mathbb{F}^n$, a v-verifier zero-knowledge proof system on secret-shared data consists of the following algorithms:

$\mathsf{Gen}(x_1, \dots x_v) \to (\pi_1, \dots, \pi_v, \pi^{\mathsf{pub}})$. Takes as input $x_i \in \mathbb{F}^n$ corresponding to every verifier $i \in [v]$ and outputs v private proofs π_i and one public proof π^{pub}.

$\mathsf{Eval}(x_i, \pi_i, \pi^{\mathsf{pub}}; r) \to \mathsf{vtag}_i \in \mathbb{F}^q$. Takes as input the i-th verifier input x_i, private proof π_i, the public proof π^{pub}, and a random tape r. Returns a proof tag vtag_i of size q.

$\mathsf{Ver}(\mathsf{vtag}_1, \dots, \mathsf{vtag}_v) \in \mathbb{F}$. Takes as input the v verification tags and checks whether to accept or reject the proof. By convention, output $0 \in \mathbb{F}$ indicates acceptance.

Silent verification. We say that the proof system is *silently verifiable* if the verification predicate Ver computes a *linear* function (over field $\mathbb{F}$) of the verification tags it takes as input. The tag size q is one and Ver checks that the (scaled) verification tags sum to zero, both follow from the linearity.

A zero-knowledge proof system on secret-shared data—whether silently verifiable or not—must satisfy the following completeness, soundness, and zero-knowledge properties.

Completeness. Completeness states that verification will always succeed if $x \in \mathcal{L}$ and all the parties are honest.

Soundness. Soundness states that a prover trying to prove that x is in the language $\mathcal{L}$ for $x \notin \mathcal{L}$ will fail the verification for v honest verifiers.

Zero knowledge. Informally, any strict subset of the verifiers does not learn any information about the prover's private input x. In our private analytics use case, this property guarantees that the client leaks no information about its private data to an adversarial coalition of up to $v-1$ out of the v servers.

Efficiency metrics. The most important efficiency metric in a silently verifiable proof system is the *proof size*—the number of bits that Gen outputs. The proof size dictates the number of bits that the prover must send to the verifiers during one interaction. Server compute is also an important efficiency metric. This quantity is largely dependent on the properties of the underlying non-silent proof system.

3.2. Features of silently verifiable proofs

We now quickly mention two useful properties of silently verifiable proofs:

Batch checking. A set of verifiers can check an arbitrarily large batch of silently verifiable proofs at the same communication cost as checking a single proof. Recall that, to verify a silently verifiable proof, the verifiers

- each compute a verification tag from their input, and
- check that their verification tags sum to zero.

To verify a batch of B proofs, the verifiers compute the verification tags for each of the B proofs as before. Rather than broadcasting the verification tags for each proof separately, the verifiers can agree on a shared random test vector $t \in \mathbb{F}^B$. Each verifier i publishes the inner product of their B verification tags (as a vector in $\mathbb{F}^B$) with the shared random vector t (Figure 2). If any set of verification tags in the batch sums to a non-zero value, then the combined verification tag will be non-zero with probability at least $1 - \frac{1}{|\mathbb{F}|}$.

Zero-knowledge against malicious verifiers. By definition, silently verifiable proof systems provide zero-knowledge even if a subset of the verifiers is malicious. In fact, this strong privacy guarantee comes for free because each verifier just sends a single message. Provided that the prover is honest, the messages that honest verifiers send are independent of the error that malicious verifiers introduce in their messages. Therefore, malicious verifiers can learn no additional information about the client's private input by deviating from the prescribed protocol.

3.3. General construction: silently verifiable proofs

To sketch how our silently verifiable proofs work, consider a prover who wants to prove that its input x lies in some

Proof system	Prover to verifier	Verifier to verifier	
		All good	d bad
Non-silent	$\lvert\pi\rvert$	pq	pq
Silent	$\lvert\pi\rvert + v + q$	1	$d \log_2 \frac{p}{d}$

TABLE 1: Communication in field elements for silently verifiable proofs and the underlying non-silent proof system. There are p provers and a small set of v verifiers. The non-silent proof system has tag size q. Entries represent the comm. from each prover to each verifier, and verifier to verifier comm. to verify the batch of p proofs. The proofs are either all honestly generated or d out of p are malicious. $O(\cdot)$ notation is suppressed for readability.

language $\mathcal{L}$. Each verifier holds a secret share of the input x. Furthermore, say that we have a zero-knowledge proof system $\Pi_{\mathcal{L}}$ on secret-shared data for the language $\mathcal{L}$ in which the verifiers communicate with each other over a broadcast channel (existing protocols satisfy this property [16], [36]).

To generate the silent proof, the prover locally simulates the execution of all of the parties (prover and verifiers) running the protocol $\Pi_{\mathcal{L}}$. The prover then sends to each verifier (1) a transcript of all messages that the simulated verifiers exchanged via the broadcast channel and (2) the view of each verifier in the simulated protocol. To check the proof, the silent verifiers only need to check that (a) their transcripts of the simulated broadcast channel are identical and (b) their simulated views are correct according to the protocol $\Pi_{\mathcal{L}}$. The verifiers can locally generate secret shares of a test value that is zero if and only if both of these checks pass (with high probability). To check a batch of proofs at once, the verifiers can publish a random linear combination of each proof's test value and accept if the resulting value sums to zero. We depict this construction in Figure 3.

For specific zero knowledge proof systems on secret shared data, we give silently verifiable proof constructions with particularly small proof sizes.

Constant-degree languages. Languages with a constant degree (typically $d = 2$) define the valid submissions for many statistics like (vector) SUM, MEAN, VARIANCE and FREQUENCY COUNT. For these languages, prior work [16] constructs and implements [39] zero-knowledge proofs on secret-shared data with proof size $O(\sqrt{M})$, where M is the number of multiplication gates in the valid predicate. Using these non-silent proof systems, we can generate silently verifiable proofs with proof size $O(\sqrt{M})$.

Language of vectors of Hamming-weight one. In private-aggregation applications, the client must often prove to the servers that it has given them secret shares of a vector of Hamming-weight one. This arises, for example, when computing the FREQUENCY COUNT and APPROXIMATE FREQUENCY statistics [32], and sketching data structures for statistics like heavy hitters (§5). Prior work on arithmetic-sketching schemes [17]–[19] gives protocols that a set of verifiers can use to test that a secret-shared vector has Hamming-weight one, while communicating only a constant number of field elements. We can compile this protocol into a silently verifiable proof.

4. Collecting Aggregate Statistics

Here, we give a short overview on how we use silently verifiable proofs to compute aggregate statistics.

4.1. Preliminaries: Additive encodings

We recall additive encodings, as used in Prio [32] and other private-aggregation systems [17], [20], [36], [44], [58], [61], [62], [67], [73]. For an input space $\mathcal{X}$, an output space $\mathcal{Y}$, and number of inputs n, let $f\colon \mathcal{X}^n \to \mathcal{Y}$ be an aggregation function. For a finite field $\mathbb{F}$ and encoding length ℓ, a private additive encoding for f consists of three efficient algorithms:

- **Encoder** $E(x) \to e$. Outputs an encoding $e \in \mathbb{F}^\ell$ of input $x \in \mathcal{X}$.
- **Verifier** $V(e) \to \{0,1\}$. Verifies an encoding $e \in \mathbb{F}^\ell$.
- **Decoder** $D(e) \to y$. Outputs the decoding $y \in \mathcal{Y}$ of its input $e \in \mathbb{F}^\ell$.

We want to use these three functions to compute f in a privacy preserving way. Intuitively, many clients will each encode their input $x_i \in \mathcal{X}$ using $E(x)$ to get e_i, and an honest client's encoding will verify under $V(e)$. We can then sum up encodings $e_1, e_2, \dots e_n$ to get a sum s, and we can run $D(s)$ on that sum to compute our original function $f(x_1, x_2, \dots x_n)$. The servers additionally learn some limited leakage from the encodings and the sum, we omit this discussion here for brevity.

4.2. Private-aggregation scheme

Building blocks. The private-aggregation protocol with n clients and v servers for the function f works over a finite field $\mathbb{F}$ and requires two building blocks:

- A private additive encoding (E, V, D) over $\mathbb{F}$ with input space $\mathcal{X}$, output space $\mathcal{Y}$ and encoding length ℓ for the aggregation function f with leakage $\hat{f}$.
- A silently verifiable proof system $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Ver})$ over $\mathbb{F}$ for the language $\mathcal{L} = \{e \mid V(e) = 1 \text{ and } e \in \mathbb{F}^\ell\}$ with v verifiers, where ℓ is the encoding length and V is the additive-encoding verifier.

Protocol. At a high level, the protocol proceeds as follows: Each client $i \in [n]$ performs the following steps:

- On input x_i, generate an additive encoding $e \leftarrow E(x_i) \in \mathbb{F}^\ell$ of the input. Split the encoding into v additive shares: $e = e_1 + \cdots + e_v \in \mathbb{F}^\ell$.
- Generate a silently verifiable proof that the encoding is well formed: $(\pi_1, \dots, \pi_v, \pi^{\mathsf{pub}}) \leftarrow \mathsf{Gen}(e_1, \dots, e_v)$.
- For each server $j \in [v]$, send (e_j, π_j) to server j. Send π^{pub} to all servers.

Next, each server $j \in [v]$ performs the following steps:

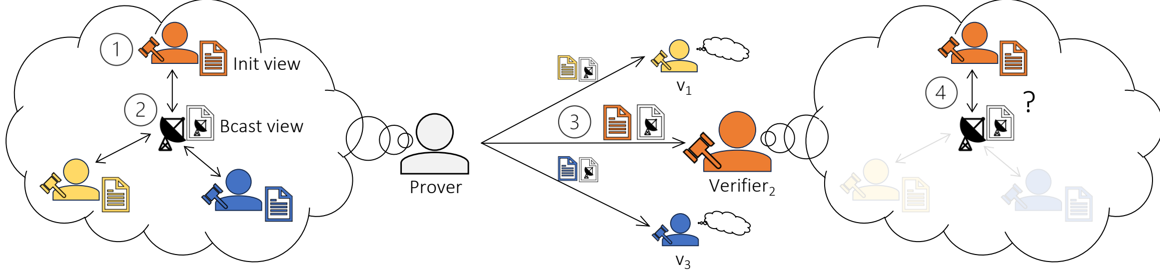

Figure 3: Overview: Constructing silently verifiable proofs from zero-knowledge proofs on secret-shared data. Given a zero-knowledge proof Π on secret-shared data, the prover, in its head, (1) initializes each verifier's view, and (2) simulates their interaction as per Π to generate the broadcast view. (3) It sends to each real verifier their initial view and the simulated broadcast view. (4) Each verifier locally verifies a part of the simulation to generate a share of the final decision.

- For each client $i \in [n]$, generate a verification tag $\mathsf{vtag}_i \in \mathbb{F}$ to verify that client i's submission is valid.
- Take a random linear combination (using randomness shared across all servers) of the n verification tags (one per client) to construct a batched verification tag $\mathsf{vtag}^\star_j \in \mathbb{F}$. Send this tag to the first server.

Finally, the servers perform the following steps:

- The first server checks that $\sum_{j \in [v]} \mathsf{vtag}^\star_j = 0 \in \mathbb{F}$ and broadcasts the result to all the other servers.
- If the check fails, the servers jointly employ group testing (§4.3) to identify the failing proofs and weed out the malformed inputs.
- Each server $j \in [v]$ adds its shares of each (valid) encoding to generate a share $e^\star_j \in \mathbb{F}^\ell$ of the sum of encodings. Server j sends $e^\star_j$ to the first server.
- The first server computes the sum $e^\star \leftarrow \sum_{j \in [v]} e^\star_j \in \mathbb{F}^\ell$ and computes the final output $\mathsf{out} \leftarrow D(e^\star) \in \mathcal{Y}$, i.e., the aggregate statistic over the clients' secret inputs.

As mentioned in §2.2, the servers in this protocol can verify the submissions in batches of size n_b each and locally aggregate their shares of passing submissions as they go. In the end, each server $j \in [v]$ sends its $e^\star_j$ to the first server.

Efficiency. The server storage while running the protocol is essentially just a vector in $\mathbb{F}^\ell$. The server-to-server communication depends on the number of malicious clients, which we discuss in §4.3.

4.3. Finding failing proofs

In our private-aggregation protocol, when malicious clients submit invalid proofs, the servers' batch-verification check fails. To identify the failing proofs with little server-to-server communication, we draw from the rich for group testing literature [40], [41]. There exist two general classes of group testing algorithms: adaptive and non-adaptive. Non-adaptive group testing algorithms perform a constant number of batch tests, and are guaranteed to catch up to a fixed number of malicious clients. Though these asymptotics seem attractive, since malicious clients can selectively upload to different servers in a given epoch, a direct application of non-adaptive methods would first acquire verifiers to reach a consensus on the contents of each batch. This would require a large amount of communication between the verifiers.

Whisper uses the generalized binary-splitting algorithm [41], [57], an adaptive group testing algorithm. Instead of using some pre-determined testing plan, adaptive group testing algorithms change their tests based on the results of previous tests. Though this requires more rounds of communication, it allows us to gracefully handle malicious clients that only upload to one server.

With a rough estimate on the upper bound of the number of "defective" uploads d in each batch of n_b clients, the servers first split the batch into d non-overlapping sub-batches of n_b/d clients each and compute $\mathsf{vtag}^\star_i$ for $i \in [v]$ for each such batch. They exchange these verification tags to find which batches contain defective uploads. For each defective batch, they binary search for defective clients within each batch in parallel. They continue this binary search until they are left with defective singleton batches—these are the malicious clients. This requires $1 + \log \frac{n_b}{d}$ rounds of server interaction and $O(vd \log \frac{n_b}{d})$ field elements in total communication per batch of n_b clients.

In order for the servers to form consistent sub-batches even in the presence of malicious clients, we leverage each client's unique id (from §2.2). During setup, the servers will share a key for a pseudorandom function, and use it to map each id to a random and deterministic sub-batch. All future splitting is done based on this PRF output, which allows for graceful detection of clients with asymmetric uploads.

5. Sketching for heavy hitters

The heavy-hitters aggregate statistic takes as input a set of n strings, each L bits long. It returns the set of strings that appear more than a certain number of times in the input. Prior work [17], [68] has proposed custom protocols for efficient computation of *exact* heavy hitters. A limitation of these protocols is that they require $O(L)$ rounds and they do not support streaming computation (i.e., the servers must store and repeatedly compute over all client submissions).

In this section, we consider the relaxed problem of computing *approximate* heavy hitters—we tolerate a small probability of failure in outputting the heavy hitters. The benefit is that we get a streaming-friendly protocol with round complexity constant in the string length L.

Our approach, following prior work on private aggregation [67], is to use linear sketches [25], [29], [30], [63], [72]. We apply a simplified version of Pagh et al's sketch [72] to approximate heavy hitters, allowing server computation to be polynomial in the string length L.

Notation. In this section, all arithmetic happens over a finite field $\mathbb{F}$ with size $|\mathbb{F}|$, which we assume to be $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ for a prime modulus p. We treat elements $\{1, \ldots, \lfloor \frac{p}{2} \rfloor\}$ as positive and $\{p-\lfloor \frac{p}{2} \rfloor, \ldots, p-1\}$ as negative. The value $-x$ represents the field element $p-x$. We use the total ordering $-\lfloor \frac{p}{2} \rfloor < \cdots < -1 < 0 < 1 < \cdots < \lfloor \frac{p}{2} \rfloor$.

5.1. Building block: Bucketed string counting

Our private-heavy-hitters construction uses the private-aggregation scheme of §4 as a subroutine. In particular, we instantiate that private-aggregation protocol with an aggregation function that we call "bucketed-string-counting."

The aggregation function is parameterized by a number of buckets B, number of client inputs n, and a string length L. Each client holds a pair of a bucket ID in $\{1, \ldots, B\}$ and an L-bit string σ. For each bucket $b \in [B]$, the aggregation function puts the "average" of the strings in bucket b. That is, if we view each string as a vector $\hat{\sigma} \in \{-1, 1\}^L \subseteq \mathbb{F}$, then for each bucket $b \in [B]$, the aggregation function sums up the values in each bucket.

Private-aggregation for bucketed string counting. We provide a simple additive encoding (E, V, D) for bucketed string counting with encoding length $\ell = (L+1) \cdot B$ and no leakage. Informally, the validity predicate ensures that the client only inserted a string into a single bucket (i.e., that there is only one bucket-aligned run of non-zero values) and that the string is encoded in $\{-1, 1\}^L$. This is instantiated using the SUM additive encoding.

We use arithmetic sketching [18], [19] to construct silently verifiable proofs for the language of valid encodings. The encoding and the proof system then, via the private-aggregation protocol of Section 4.2, yield a private-aggregation scheme for bucketed string counting.

Optimization in the two-server case. We spent a large portion of this semester trying to apply these arithmetic sketches to heavy hitters. The sketch of [19] turned out to be our most promising candidate. Later improvements to their sketch such as [18] optimized for server-server communication, but silently verifiable proofs dramatically reduce this cost regardless of the sketch details. Ultimately, verifiable distributed point functions (VDPFs) [19], [37] proved more optimal for the two server case. A verifiable DPF gives a succinct way to secret share a weight-one vector. The verifiability property means that two servers, each holding a purported succinct share of a weight-one vector, can tell that their shares are well formed by performing an equality check on a short string. As with our silently verifiable proofs, it is possible for the servers to batch-verify a large number of VDPFs by exchanging a short string. VDPFs thus can replace silently verifiable proofs in the two-server setting for heavy hitters.

At a high level, a VDPF's concrete efficiency comes from its use of AES hardware instructions. These kinds of optimizations are not available when working with finite fields, though there is some future work to explore SIMD instructions for finite field operations.

Heavy hitters protocol. The scheme is parameterized by a number of clients n, a string length L, a number of buckets B, a hash function $H_{\text{bucket}}\colon \{0,1\}^L \to [B]$, a hash function $H_{\text{sign}}\colon \{0,1\}^L \to \{0,1\}$, and a heavy-hitter threshold T.

Client input preparation. Given a string $x \in \{0,1\}^L$ as input:

- The client hashes the string to get a bucket ID b and sign bit β:

$$b \leftarrow H_{\text{bucket}}(x) \in [B] \quad \text{and} \quad \beta \leftarrow H_{\text{sign}}(x) \in \{0,1\}.$$

- If $\beta = 0$, the client complements its bitstring $x \leftarrow \bar{x}$.
- The client participates in the secure-aggregation protocol for bucketed string counting using input $(b, \beta \| x) \in [B] \times \{0,1\}^{L+1}$.

Output decoding. The output of the secure-aggregation protocol is, for each bucket, (1) the number of strings in that bucket and (2) the sum over $\mathbb{F}^{L+1}$ of all strings in that bucket. This output-decoding procedure recovers the set of approximate heavy hitters from this output.

Initialize a set $H = \emptyset$ of heavy hitters. Then, for each bucket $b \in [B]$ containing at least T strings:

- Let $s \in \mathbb{F}^{L+1}$ be the sum of the strings in bucket b.
- "Round" s to a bitstring $\hat{\sigma} \in \{0,1\}^{L+1}$ by mapping each value in $\{-n, \ldots, 0\} \subseteq \mathbb{F}$ to 0 and all other values to 1.
- Parse $(\beta, \sigma) \leftarrow \hat{\sigma} \in \{0,1\} \times \{0,1\}^L$.
- If $\beta = 0$, complement the bits of σ: $\sigma \leftarrow \bar{\sigma}$.
- Add σ to the set of heavy hitters H.

Finally, output H as the set of heavy hitters.

Figure 4: Our protocol for approximate heavy hitters.

5.2. Our heavy-hitters protocol

In our protocol (Figure 4), each client first hashes its input string $x \in \{0,1\}^L$ into one of B buckets, where B is a protocol parameter. The client and servers then run the private-aggregation protocol for bucketed string counting to compute the "average" of the strings in each bucket.

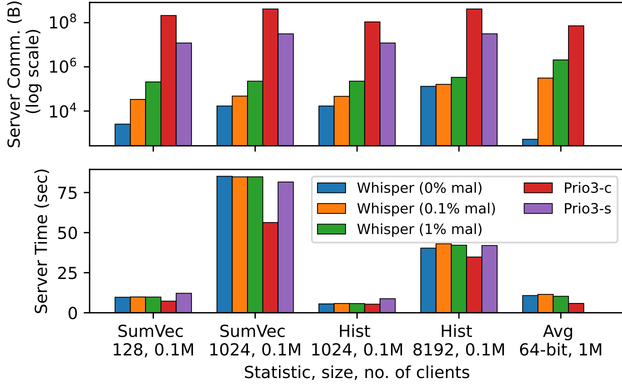

Figure 5: Server-to-server communication and time of each server for verification and aggregation of common statistics.

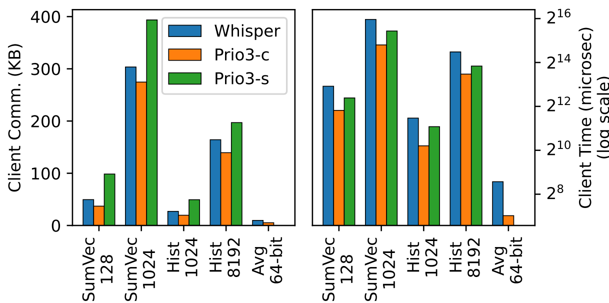

Figure 6: Communication and proof generation time per client for common statistics.

If there were no collisions—i.e., if two clients have distinct strings they hash to distinct buckets—then the output of the bucketed string counting function would exactly give the set of all heavy hitters. However, since multiple distinct strings may fall into the same bucket, we need to recover heavy hitters despite collisions.

The delicate part of the analysis is showing that, for the purposes of finding heavy hitters, these collisions do not matter too much. That is, if a string is a heavy hitter, it is unlikely that it will fall into a bucket containing so many non-heavy-hitters that we cannot recover the original heavy hitter. Not only do we need to consider honest collisions, we also need to consider malicious clients who deliberately choose to upload strings that collide with heavy hitters, preventing the true heavy hitter from being recovered. This analysis is omitted for brevity. To recover a heavy hitter from a bucket, we just round each bit of the bucket's counter either up or down to determine whether the corresponding bit of the string is either 0 or 1.

6. Evaluation

We implement Whisper in Rust on top of the `libprio-rs` library [39]. Implementations of both Whisper and the comparison systems are multithreaded. For heavy hitters, we implement our two-server optimization (§5) that uses VDPFs

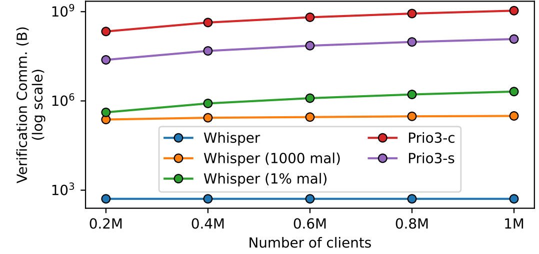

Figure 7: Verification communication per server with an increasing number of clients for Hist 1024.

and given their compatibility with rings $\mathbb{Z}_{2^k}$ for $k \in \mathbb{N}$, our heavy hitters code runs over $\mathbb{Z}_{2^{16}}$ and $\mathbb{Z}_{2^{32}}$ (depending on the number of clients) for faster arithmetic. We use SHA-256 to batch multiple verification tags for VDPFs. Our VDPF code borrows from Poplar's codebase [31]. To be sound against adversaries that run in time at most $\approx 2^{128}$, for general statistics, we perform two parallel runs with 128-bit field each, and for heavy hitters, we set $\lambda = 128$ for VDPFs.

Evaluation setup. We use two servers to mirror existing deployments [3], [7], [52], [69]: one in Iowa (`us-central1-a`) and the other in Virginia (`us-east4-c`). Both have 32 vCPUs and 64 GB memory. We use a 2021 MacBook Pro as a client.

For our exposure notification benchmarks, we additionally instantiate Google Cloud Storage buckets local to both of these locations to store client submissions.

Metrics of success
Silently verifiable proofs reduce server-server communication, while increasing client-server communication and, in the general case, server compute. To illustrate this point, we measure these quantities, and to show that this tradeoff is often worthwhile, we additionally measure the dollar cost of running a private analytics service using our framework, using common cloud pricing models.

6.1. General statistics

Baseline. We first compare with the state-of-the-art system Prio3 [39], [58]. For some statistics, Prio3 has a "chunk-size" parameter that trades client-to-server communication for server-to-server communication. We call the client-optimized configuration *Prio3-c* and the server-optimized configuration *Prio3-s*. We compare with both.

Statistics. We consider three main statistics supported by Prio3: VECTOR SUM ("sumvec"), FREQUENCY COUNT ("hist"), and MEAN ("avg"). For vector sums, we consider vectors of size 128 and 1024, and 16-bit entries. For frequency count, we consider 1024 and 8192 bins. We compute means over 64-bit values.

Server performance. Figure 5 shows the server-to-server communication and server time (after submissions are received) for Whisper and Prio3. Increasing the number of malicious clients barely affects Whisper's server time. However, as we discuss in §4.3, finding d malicious clients

requires communication $O(d \log \frac{n}{d})$, and therefore, server communication increases as the number of malicious clients increases. The server-to-server communication remains up to three orders of magnitude lower than the Prio3-c baseline. The communication-cost improvement comes at an average cost of roughly a 1.4× increase in server time.

Client performance. Figure 6 compares Whisper with Prio3 on client communication (encoding + proof size) and client time (proof generation). Whisper has roughly 1.4× more client communication than Prio3-c. As the size of the statistics increases, the increase in our client communication relative to Prio3 goes down. Our client time is at most a few milliseconds and about 2-3× higher than baseline.

Server-optimized Prio3. Whisper improves server-to-server communication by up to two orders of magnitude over Prio3-s. Whisper outperforms Prio3-s in *both* client and server communication, and the server time is comparable.

Dollar cost. Using Google Cloud's pricing model [50], [53], we estimate up to 3× reduction in the cost of running the servers (about 2× reduction on average) over our baseline.

Silently verifiable proofs. Figure 7 shows batch verifiability of our silently verifiable proofs. When the number of malicious clients is fixed, verification communication stays constant as clients increase. Prio3's proof verification communication scales linearly with clients. Our batch verification comes at some increase in proof size, proof generation time, and proof verification time (Figures 5 and 6).

6.2. Heavy hitters

Baseline. We compare with Poplar [17], the state-of-the-art system for private heavy hitters in the two-server setting.

Parameters. We sample 256-bit client inputs from a Zipf distribution with parameter 1.03 and support 10,000, as in Poplar's evaluation [17]. We configure Poplar as in their evaluation. For Whisper, we set the parameters such that the probability of finding all the heavy hitters is at least 0.999. We consider two heavy hitter thresholds 1% and 0.1% of the total number of clients. When using the 1% threshold, we use a sketch with 256 buckets and 14 sketching instances. When using the 0.1% threshold, we use 1024 buckets and 17 sketching instances.

We set the number of malicious clients as half the heavy hitter threshold, and to maintain our success probability, we double the number of buckets in our experiments with malicious clients. For our streaming experiment, we form batches of 3,000 clients. Each batch is verified, performing group testing if necessary to sanitize malicious clients, and then aggregated into the small heavy hitters sketch before processing the next batch.

Streaming. Figure 8 shows server runtime to process large streams with millions of clients. Poplar cannot stream its computation and keeps all submissions in memory. At around 1.5M clients, its memory usage exceeds the server's

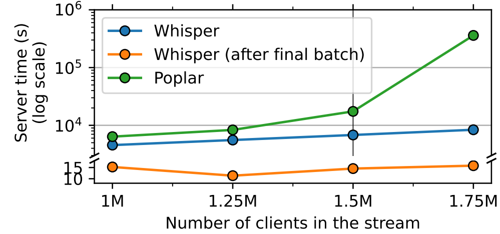

Figure 8: Server time to compute heavy hitters over a stream of client submissions for 0.1% threshold and 0.05% malicious clients. Poplar runs out of the main memory at the vertical line.

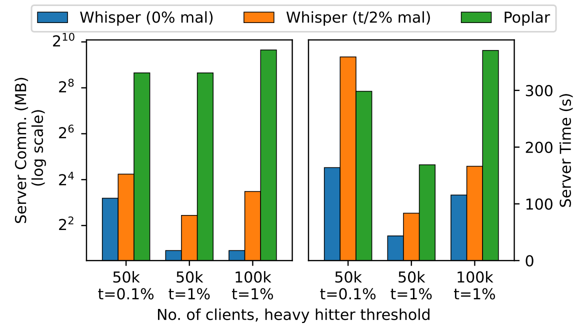

Figure 9: Server-to-server communication and time of each server to compute heavy hitters.

memory, and swapping to disk degrades the system performance. Mitigating this slowdown would require using larger, more expensive servers. Whisper uses streaming to avoid this slowdown. Moreover, with the fixed batch size, Whisper's server time after the last submission is independent of the stream size.

Other metrics. Whisper's server time is lower than Poplar in most cases (Figure 9) and server-to-server communication is lower by one to two orders of magnitude (Figure 9). This translates to up to 3.8× reduction in the dollar cost to run the servers based on Google Cloud's pricing model [50], [53]. However, the client communication in Whisper is 14-17× larger than in Poplar, and the client is around 2× slower. In absolute terms, our client communication is less than 500 KB for both the heavy-hitter thresholds. Moreover, historically, cloud providers don't charge for ingress communication. When the probability of heavy-hitter recovery is 0.9, client communication is 7-10× higher than Poplar.

6.3. Exposure Notifications benchmark

To better understand our system's performance in a real world deployment, we measure our performance with a realistic suite of prio-style aggregate statistics, used in 2020 for Apple and Google's exposure notification private analytics application [52]. We simulate 1 million client uploads into two Google Cloud Storage Buckets, located in the same physical regions as their associated processing servers. We

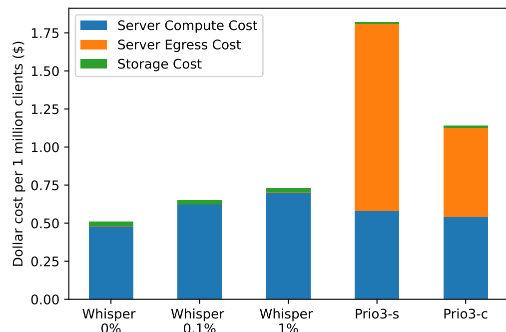

Figure 10: Dollar cost breakdown for ENPA aggregate statistics over 1 million clients

evaluate Whisper with 0, 0.1, and 1% simulated malicious clients. We do not distinguish between different malicious behaviors – all client deviations from the protocol that we could think of result in the same kind of detection, and the same effect on the eventual group testing pattern.

Workload In 2020, Google and Apple collaborated to collect private aggregate statistics over mobile phone users, in order to document the spread of Covid. Google's open source code [51] collects 14 different statistics, concerning dates of exposures, vaccination status, and frequency of exposures. Most of these were expressed as Prio3 Histograms, containing less than 100 buckets. The vast majority of the computation was spent computing a single histogram of 1152 buckets.

Server Performance Similar to the Prio3 benchmarks, Whisper's server-server communication was 2-5 orders of magnitude lower than the base implementation. This high communication proved to bottleneck server performance as well – relative to the baseline, our code was much faster for this workload than the simple Prio3 workloads, having at most about a 15% difference. Though Whisper's submissions were 2x larger than Prio3's, the additional storage cost was almost negligible. We instantiated the upload service using a 512MB memory / 0.33 vCPU Google Cloud Function. The cost of running this service was the same for both Whisper and Prio3 approaches. Overall, we see between 1.5x-3.5x cost savings.

7. Related Work

Single-server model. There is a rich literature on systems for private collection of aggregate statistics via local differential privacy [4], [10], [11], [24], [67], often using sketching algorithms as Whisper does. These systems provide an incomparable privacy property to Whisper: we aim for an MPC-style privacy property—nothing leaks beyond the aggregate statistic—while these systems provide user-level differential privacy (and they do leak information about each user's data beyond the aggregate statistic itself). A secondary distinction is that these systems do not necessarily protect correctness against malicious clients [4], [10], [11], [15], [24], [28], [42], [60], [67], [78]; adding such defenses can be expensive [12], [15], [44], [62], [65]–[67], [70].

Private heavy hitters. Mix-net [26], [71] and other anonymous communication systems [33], [46], [47] can be used to compute heavy hitters from the multiset of clients' strings while providing anonymity, however, the entire multiset of the client inputs leaks in the process. In the distributed-trust setting, existing protocols incur high server-to-server communication [8], [14], [59], cannot compute heavy hitters over a stream leading to large server-side storage [68] or both [17]. Star [35] considers a different setting with an aggregation server with a separate randomness server and doesn't hide the identity of clients with the same input. Except for Plasma [68], the server egress in all these works scales linearly with the number of clients. Plasma works in a different threat model than Whisper assuming an honest majority among three servers which can be challenging to find in the real-world [64]. Moreover, Plasma and the two-server state-of-the-art Poplar [17] cannot stream heavy hitter computation leading to large server storage and require the servers to interact over multiple rounds.

Differentially private aggregate statistics. There is a long line of work [4], [10], [11], [21], [23], [24], [45], [74], [77] on computing aggregate statistics over randomized responses collected from the clients. The noise added by the clients provides differential privacy, however, it leads to a loss in the accuracy of the output and makes it challenging to filter malformed submissions. Moreover, noisy submissions from each client don't completely hide all private information. Whisper and related systems [2], [17], [32], [68] provide a different privacy guarantee where only the output and a modest leakage function are seen by the servers, and the accuracy of the output is preserved. However when the leakage from the output is a concern, Whisper can easily be extended to use differential privacy where, similar to [17], [27], [32], [67], the noise is added directly to the aggregate [76]. This maintains higher accuracy compared to local differential privacy. Zhu et al. [28], [78] develop a trie-based heavy hitters protocol where subsampling the clients provides meaningful differential privacy without requiring additional noise. Prochlo [13] requires a trusted shuffler.

Batch verifiable proofs on secret-shared data. Zero-knowledge proofs on secret-shared data supporting batch verification are implicit in recent work by Hazay et al. [55] where the proof sizes are at least linear in the size of the predicate. Our silently verifiable proofs provide batch verification with sublinear-sized proofs for structured languages common in private analytics. For the language of one-hot vectors, verifiable distributed point functions [37] offer batch verification and succinct key sizes.

References

[1] "Mozilla Telemetry Data Documentation: Raw Ping Data, Ping Types," https://docs.telemetry.mozilla.org/datasets/pings.html, [Online; accessed Oct-2023].

[2] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, "Prio+: Privacy preserving aggregate statistics via boolean shares," in *SCN*, 2022.

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

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

[5] ——, "Managed device attestation for apple devices," https://support.apple.com/guide/deployment/managed-device-attestation-dep28afbde6a/web.

[6] ——, "Mitigate fraud with AppAttest and DeviceCheck," *WWDC21*, 2021.

[7] Apple and Google, "Exposure notification privacy-preserving analytics (enpa) white paper," https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[8] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, "Efficient secure three-party sorting with applications to data analysis and heavy hitters," in *CCS*, 2022.

[9] L. Bangalore, M. H. F. Sereshgi, C. Hazay, and M. Venkitasubramaniam, "Flag: A framework for lightweight robust secure aggregation," in *AsiaCCS*, 2023.

[10] R. Bassily, K. Nissim, U. Stemmer, and A. G. Thakurta, "Practical locally private heavy hitters," in *NIPS*, 2017.

[11] R. Bassily and A. D. Smith, "Local, private, efficient protocols for succinct histograms," in *STOC*, 2015.

[12] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun, "ACORN: input validation for secure aggregation," in *USENIX Security Symposium*, 2023.

[13] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld, "Prochlo: Strong privacy for analytics in the crowd," in *SOSP*, 2017.

[14] J. Böhler and F. Kerschbaum, "Secure multi-party computation of differentially private heavy hitters," in *CCS*, 2021.

[15] K. A. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *CCS*, 2017.

[16] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear pcps," in *CRYPTO (3)*, 2019.

[17] ——, "Lightweight techniques for private heavy hitters," in *IEEE Symposium on Security and Privacy*, 2021.

[18] ——, "Arithmetic sketching," in *CRYPTO (1)*, 2023.

[19] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *CCS*, 2016.

[20] A. Broadbent and A. Tapp, "Information-theoretic security without an honest majority," in *ASIACRYPT*, 2007.

[21] M. Bun, J. Nelson, and U. Stemmer, "Heavy hitters and the structure of local privacy," *ACM Trans. Algorithms*, vol. 15, 2019.

[22] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA:privacy-preserving aggregation of multi-domain network events and statistics," in *USENIX Security*, 2010.

[23] K. N. Chadha, J. Chen, J. C. Duchi, V. Feldman, H. Hashemi, O. Javidbakht, A. McMillan, and K. Talwar, "Differentially private heavy hitter detection using federated analytics," *CoRR*, vol. abs/2307.11749, 2023.

[24] T. H. Chan, M. Li, E. Shi, and W. Xu, "Differentially private continual monitoring of heavy hitters from distributed streams," in *Privacy Enhancing Technologies*, 2012.

[25] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theor. Comput. Sci.*, vol. 312, 2004.

[26] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, 1981.

[27] S. G. Choi, D. Dachman-Soled, M. Kulkarni, and A. Yerukhimovich, "Differentially-private multi-party sketching for large-scale statistics," *Proc. Priv. Enhancing Technol.*, 2020.

[28] G. Cormode and A. Bharadwaj, "Sample-and-threshold differential privacy: Histograms and applications," in *AISTATS*, 2022.

[29] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proc. VLDB Endow.*, 2008.

[30] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, 2005.

[31] H. Corrigan-Gibbs, "heavyhitters," https://github.com/henrycg/heavyhitters.

[32] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *NSDI*, 2017.

[33] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An anonymous messaging system handling millions of users," in *IEEE Symposium on Security and Privacy*, 2015.

[34] G. Danezis, C. Fournet, M. Kohlweiss, and S. Z. Béguelin, "Smart meter aggregation via secret-sharing," in *SEGS@CCS*, 2013.

[35] A. Davidson, P. Snyder, E. B. Quirk, J. Genereux, B. Livshits, and H. Haddadi, "STAR: secret sharing for private threshold aggregation reporting," in *CCS*, 2022.

[36] H. Davis, C. Patton, M. Rosulek, and P. Schoppmann, "Verifiable distributed aggregation functions," *Proc. Priv. Enhancing Technol.*, 2023.

[37] L. de Castro and A. Polychroniadou, "Lightweight, maliciously secure verifiable function secret sharing," in *EUROCRYPT (1)*, 2022.

[38] Divvi Up, https://divviup.org/.

[39] ——, https://github.com/divviup/libprio-rs.

[40] R. Dorfman, "The Detection of Defective Members of Large Populations," *The Annals of Mathematical Statistics*, vol. 14, no. 4, 1943.

[41] D. Du, F. K. Hwang, and F. Hwang, *Combinatorial group testing and its applications*. World Scientific, 2000, vol. 12.

[42] Y. Duan, N. Youdao, J. F. Canny, and J. Z. Zhan, "P4P: practical large-scale privacy-preserving distributed computation robust against malicious users," in *USENIX Security Symposium*, 2010.

[43] C. Dwork, "Differential privacy: A survey of results," in *TAMC*, 2008.

[44] T. Elahi, G. Danezis, and I. Goldberg, "Privex: Private collection of traffic statistics for anonymous communication networks," in *CCS*, 2014.

[45] Ú. Erlingsson, V. Pihur, and A. Korolova, "RAPPOR: randomized aggregatable privacy-preserving ordinal response," in *CCS*, 2014.

[46] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," in *NDSS*, 2022.

[47] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *USENIX Security Symposium*, 2021.

[48] M. Faisal, J. Zhang, J. Liagouris, V. Kalavri, and M. Varia, "TVA: A multi-party computation system for secure and expressive time series analytics," in *USENIX Security Symposium*, 2023.

[49] D. Froelicher, J. R. Troncoso-Pastoriza, J. S. Sousa, and J. Hubaux, "Drynx: Decentralized, secure, verifiable system for statistical queries and machine learning on distributed datasets," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, 2020.

[50] Google, "All networking pricing," https://cloud.google.com/vpc/network-pricing.

[51] ——, "exposure-notifications-android," https://github.com/google/exposure-notifications-android/.

[52] ——, “Exposure notifications: Help slow the spread of covid-19, with one step on your phone,” "https://www.google.com/covid19/exposurenotifications".

[53] ——, “VM instance pricing,” https://cloud.google.com/compute/vm-instance-pricing.

[54] S. Halevi, Y. Ishai, E. Kushilevitz, and T. Rabin, “Additive randomized encodings and their applications,” in *CRYPTO (1)*, 2023.

[55] C. Hazay, M. Venkitasubramaniam, and M. Weiss, “Your reputation’s safe with me: Framing-free distributed zero-knowledge proofs,” *IACR Cryptol. ePrint Arch.*, p. 1523, 2022.

[56] T. Humphries, R. A. Mahdavi, S. Veitch, and F. Kerschbaum, “Selective MPC: distributed computation of differentially private key-value statistics,” in *CCS*, 2022.

[57] F. K. Hwang, “A method for detecting all defective members in a population by group testing,” *Journal of the American Statistical Association*, 1972.

[58] IETF, “Verifiable distributed aggregation functions draft-irtf-cfrg-vdaf-07,” https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/.

[59] P. Jangir, N. Koti, V. B. Kukkala, A. Patra, B. R. Gopal, and S. Sangal, “Poster: Vogue: Faster computation of private heavy hitters,” in *CCS*, 2022.

[60] M. Jawurek and F. Kerschbaum, “Fault-tolerant privacy-preserving statistics,” in *Privacy Enhancing Technologies*, 2012.

[61] A. F. Karr, X. Lin, A. P. Sanil, and J. P. Reiter, “Regression on distributed databases via secure multi-party computation,” in *DG.O*, 2004.

[62] K. Kursawe, G. Danezis, and M. Kohlweiss, “Privacy-friendly aggregation for the smart-grid,” in *PETS*, 2011.

[63] K. G. Larsen, J. Nelson, H. L. Nguyen, and M. Thorup, “Heavy hitters via cluster-preserving clustering,” in *FOCS*, 2016.

[64] Y. Lindell, D. Cook, T. Geoghegan, S. Gran, R. Schmidt, E. Kret, D. Kaviani, and R. A. Popa, “The deployment dilemma: Merits & challenges of deploying mpc,” https://mpc.cs.berkeley.edu/blog/deployment-dilemma.

[65] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi, “Rofl: Robustness of secure federated learning,” in *SP*, 2023.

[66] E. Margolin, K. Newatia, T. Luo, E. Roth, and A. Haeberlen, “Arboretum: A planner for large-scale federated analytics with differential privacy,” in *SOSP*, 2023.

[67] L. Melis, G. Danezis, and E. D. Cristofaro, “Efficient private statistics with succinct sketches,” in *NDSS*, 2016.

[68] D. Mouris, P. Sarkar, and N. G. Tsoutsos, “Plasma: Private, lightweight aggregated statistics against malicious adversaries with full security,” Cryptology ePrint Archive, Paper 2023/080.

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

[70] M. Naor, B. Pinkas, and E. Ronen, “How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior,” in *CCS*, 2019.

[71] C. A. Neff, “A verifiable secret shuffle and its application to e-voting,” in *CCS*, 2001.

[72] R. Pagh, “Compressed matrix multiplication,” *ACM Trans. Comput. Theory*, vol. 5, 2013.

[73] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li, “Privacy and accountability for location-based aggregate statistics,” in *CCS*, 2011.

[74] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren, “Heavy hitter estimation over set-valued data with local differential privacy,” in *CCS*, 2016.

[75] M. Rosenberg, J. White, C. Garman, and I. Miers, “zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure,” Cryptology ePrint Archive, Paper 2022/878.

[76] T. Steinke, “Multi-central differential privacy,” *CoRR*, vol. abs/2009.05401, 2020.

[77] M. Zhou, T. Wang, T. H. Chan, G. Fanti, and E. Shi, “Locally differentially private sparse vector aggregation,” in *SP*, 2022.

[78] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li, “Federated heavy hitters discovery with differential privacy,” in *AISTATS*, 2020.