Scaling Zero Knowledge Proofs Through Application and Proof System Co-Design

Yuwen Zhang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-32
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-32.html

May 1, 2025

Scaling Zero Knowledge Proofs Through Application and Proof System Co-Design

by Yuwen Zhang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

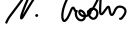
Committee:

Professor Raluca Ada Popa
Research Advisor

May 17, 2024

(Date)

* * * * * * *



Professor Natacha Crooks
Second Reader

May 17, 2024

(Date)

Scaling Zero Knowledge Proofs Through Application and Proof System Co-Design

by

Yuwen Zhang

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of

Master of Science, Plan II

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee

Professor Raluca Ada Popa, Research Advisor
Professor Natacha Crooks, Second Reader

Spring 2024

Abstract

Scaling Zero Knowledge Proofs Through Application and Proof System Co-Design

by

Yuwen Zhang

Master of Science, Plan II in Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Research Advisor

Professor Natacha Crooks, Second Reader

Zero knowledge succinct non-interactive arguments of knowledge (zkSNARKs) allow an untrusted prover to cryptographically prove that a certain statement is true without compromising their privacy. Though powerful, many existing applications of zkSNARKs do not scale for larger systems. By tailoring protocol and system design for specific use cases, I demonstrate that systems using zkSNARKs can scale well in many dimensions. In my first chapter, I focus on privacy-preserving analytics systems. Existing deployments use a small set of non-colluding servers alongside some specialized zkSNARK constructions in order to compute aggregate statistics over client data without learning any individual's information. Our system, Whisper, improves upon prior work by drastically reducing inter-server communication at the cost of slightly larger client proofs, resulting in large dollar cost savings. In my second chapter, I discuss techniques for delegated proof generation for complex circuits. In particular, I focus on the delegated prover environment, where a trusted delegator outsources proof generation to third party workers. Existing solutions either trust these workers with their secret inputs in plaintext, or they fail to fully take advantage of worker parallelism. Our system, DFS, achieves state of the art scaling without trusting workers with sensitive delegator secrets.

Acknowledgments

Thanks to Raluca Ada Popa, my advisor, for her thoughtful advice and support. I would also like to thank Mayank Rathee, for welcoming into the Sky Security group and mentoring me through my work on Whisper. Lastly, I'd like to thank Yuncong Hu and Pratyush Mishra for their leadership and guidance through my work on DFS.

Chapter 1

Introduction

Zero Knowledge Succinct Non-interactive Proofs of Knowledge (zkSNARKs) are a powerful cryptographic tool, garnering significant interest from both academia and industry. A *prover*, who holds some secret witness, presents a cryptographic argument this witness satisfies some computation. This argument is compiled into a short proof, which any *verifier* is able to efficiently verify.

In order to instantiate an efficient prover, some systems only support proof generation for very specialized computations. In Whisper, we focus on zkSNARKs in the context of privacy-preserving collection of aggregate statistics. Like prior private analytics 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. Prior systems required the servers to exchange a few bits of information to verify the well-formedness of each client submission. 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. We achieve this using *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. 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%.

Another popular research direction has been generating zkSNARKs for more general computations, allowing for a myriad of popular applications. However, despite recent advancements, the computational cost of proof generation remains extremely high. Prior systems attempt to alleviate these costs by distributing off-the-shelf zkSNARK proof generation among many workers. These works fall under two main categories: public delegation, where workers all see the witness in the clear, and private delegation, where workers need to do some slow multi-party computation (MPC) in order to compute the proof. In DFS, we create a custom zkSNARK designed with both private and public delegation in mind. By carefully choosing appropriate subprotocols, we achieve graceful scaling in the number of workers in both the public and private delegation settings.

Chapter 2

Whisper

2.1 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 [4], public health trends [8, 64], and device telemetry [87] at million-user scale.

In this paper, we focus on private-aggregation systems based on multi-party computation techniques [3, 11, 19, 41, 43, 45, 47, 53, 57, 59, 71, 77, 85, 86]. 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 [4], Google [8], Mozilla [87], and others [47] 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 [3, 11, 41, 45, 47, 71] 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. This communication cost is significant when supporting millions of clients.

More recent systems [19, 73, 86] 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 distributed-trust based systems [19, 73, 86] require the servers' secret state to grow 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.

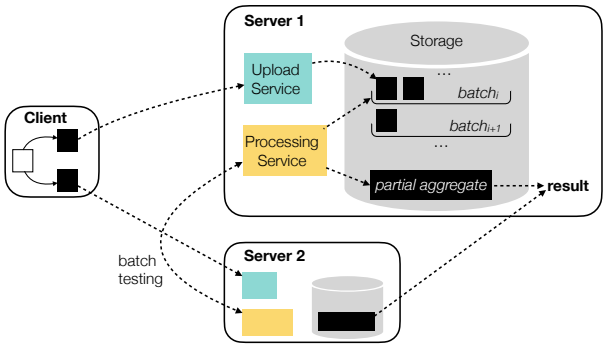


Figure 2.1: Whisper's server architecture. Clients split their data using a secret-sharing scheme and send one share to each server. The servers process submissions in batches.

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 [41, 45, 73] (Figure 2.1) 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 [18]. 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 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 [91]. 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 [48]. 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 [48]. 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 [19] 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.

My Contribution

I worked on implementing and evaluating the entire system. I also worked on designing our heavy hitters solution and choosing an efficient, correct protocol for identifying malicious clients.

2.2 System Overview

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

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 Whispercommunicate with each of the Whisperservers and do not communicate with other clients. We assume that the clients have a means to authenticate to the servers [6].

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 2.1.

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}.

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 [50]; 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.

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 [19,41,45], Whispersupports any aggregate statistic that can be computed via a verifiable *additive encoding* of the client's data [41, 69]. We discuss additive encodings in more detail in §2.4. Using encoding techniques from prior work [26, 39, 41, 53, 85, 93], Whispersupports 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) §2.5.

In many cases, Whisper reveals to the servers slightly more information about the client inputs $(x_1, \ldots, x_n)$ than the aggregate statistic $f(x_1, \ldots, x_n)$ itself. For example, for SUM, there is no extra leakage, while private-aggregation schemes for VARIANCE additionally leak the mean for efficiency [41]. In this case, as in prior work, we define the *leakage* $\hat{f}(x_1, \ldots, x_n)$ 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.

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 2.4 and 2.5; 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 $x_1, \ldots, x_n$, except what can be inferred from the aggregate statistic $f(x_1, \ldots, x_n)$ and the leakage function $\hat{f}(x_1, \ldots, x_n)$. 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.

For privacy, it is important that "enough" honest clients participate in each epoch. This ensures that $f(x_1, \ldots, x_n)$ and $\hat{f}(x_1, \ldots, x_n)$ 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 [52, 85] gives some protection in this case. To limit the number of malicious clients, as in prior works [18, 19, 41, 85], we assume that the servers employ Sybil-protection mechanisms [6, 7, 47, 95].

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.

2.3 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.

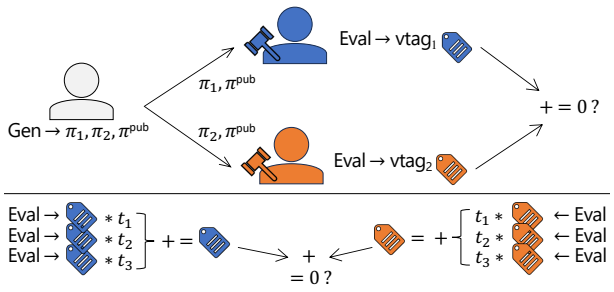


Figure 2.2: Silently verifiable proofs with batch verification.

We first recall the definition of a zero-knowledge proof on secret-shared data [18]. 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 [18, 41, 45]. 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 §2.3, silently verifiable proofs allow verifiers to check a large batch of proofs at once, with minimal verifier-to-verifier communication.

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 [18, 41, 45]. 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 $\{1, \ldots, n\}$.

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:

$\mathsf{Gen}(x_1, \ldots x_v) \to (\pi_1, \ldots, \pi_v, \pi^{\mathsf{pub}})$. 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.

$\mathsf{Ver}(\mathsf{vtag}_1, \ldots, \mathsf{vtag}_v) \in \mathbb{F}$. 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.

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 2.2). 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.

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 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 [18, 45]).

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 2.3.

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.

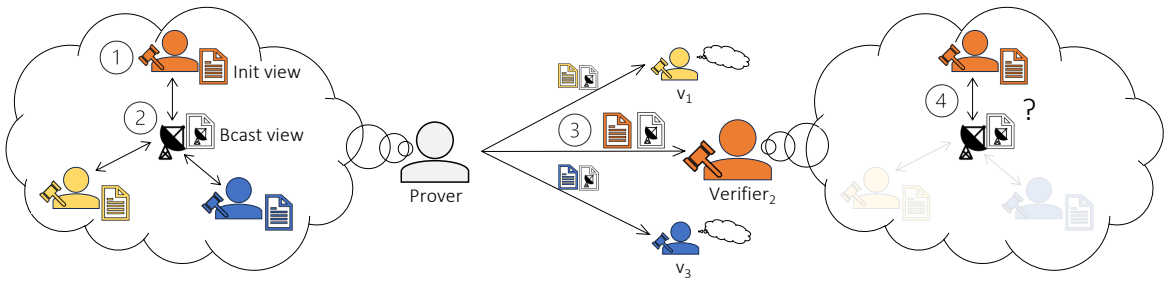


Figure 2.3: Overview: Constructing silently verifiable proofs from zero-knowledge proofs on secret-shared data. Given a zero-knowledge proof Π on secret-shared data, the prover, in its head, ① initializes each verifier's view, and ② simulates their interaction as per Π to generate the broadcast view. 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.

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

Table 2.1: 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.

For these languages, prior work [18] constructs and implements [48] 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 [41], and sketching data structures for statistics like heavy hitters (§2.5). Prior work on arithmetic-sketching schemes [19, 20, 23] 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.

2.4 Collecting Aggregate Statistics

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

Preliminaries: Additive encodings

We recall additive encodings, as used in Prio [41] and other private-aggregation systems [19, 24, 45, 53, 73, 76, 77, 85, 93]. 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 $e_1, e_2, \ldots e_n$ to get a sum s, and we can run $D(s)$ on that sum to compute our original function $f(x_1, x_2, \ldots x_n)$. The servers additionally learn some limited leakage from the encodings and the sum, we omit this discussion here for brevity.

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: $(\pi_1, \ldots, \pi_v, \pi^{\mathsf{pub}}) \leftarrow \mathsf{Gen}(e_1, \ldots, e_v)$.
- 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:

- 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 $\mathsf{vtag}_j^\star \in \mathbb{F}$. Send this tag to the first server.

Finally, the servers perform the following steps:

- The first server checks that $\sum_{j\in[v]} \mathsf{vtag}_j^\star = 0 \in \mathbb{F}$ and broadcasts the result to all the other servers.
- If the check fails, the servers jointly employ group testing (§2.4) 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 $e_j^\star \in \mathbb{F}^\ell$ of the sum of encodings. Server j sends $e_j^\star$ to the first server.
- The first server computes the sum $e^\star \leftarrow \sum_{j\in[v]} e_j^\star \in \mathbb{F}^\ell$ 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 $e_j^\star$ 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 §2.4.

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 [49,50]. 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 [50, 72], 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 $\mathsf{vtag}_i^\star$ 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.

2.5 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 [19, 86] 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 [85], is to use linear sketches [30, 38, 39, 79, 91]. We apply a simplified version of Pagh et al's sketch [91] 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 $-\lfloor \frac{p}{2} \rfloor < \cdots < -1 < 0 < 1 < \cdots < \lfloor \frac{p}{2} \rfloor$.

Building block: Bucketed string counting

Our private-heavy-hitters construction uses the private-aggregation scheme of §2.4 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 [20, 23] 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 2.4, yield a private-aggregation scheme for bucketed string counting.

Optimization in the two-server case. Though this direct application of arithmetic sketching yields a correct solution, we ultimately use an alternative construction, verifiable distributed point functions (VDPFs) [23, 46] in our implementation. 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. Our sketch requires finite field operations, which are not natively supported on hardware. 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.

Our heavy-hitters protocol

In our protocol (Figure 2.4), 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.

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.

2.6 Evaluation

We implement Whisper in Rust on top of the `libprio-rs` library [48]. Implementations of both Whisper and the comparison systems are multithreaded. For heavy hitters, we implement our two-server optimization (§2.5) that uses VDPFs 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

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 2.4: Our protocol for approximate heavy hitters.

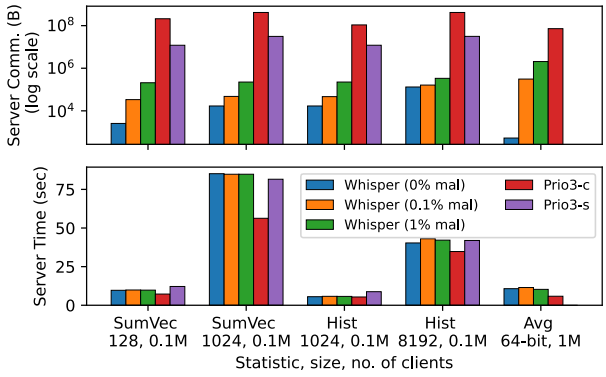


Figure 2.5: Server-to-server communication and time of each server for verification and aggregation of common statistics.

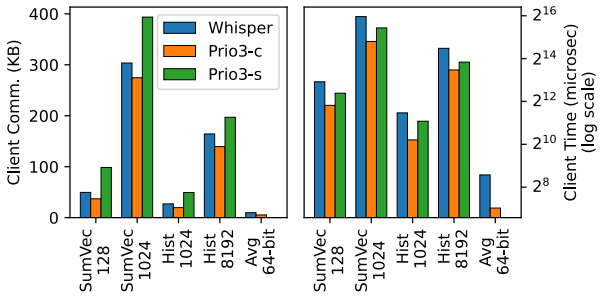


Figure 2.6: Communication and proof generation time per client for common statistics.

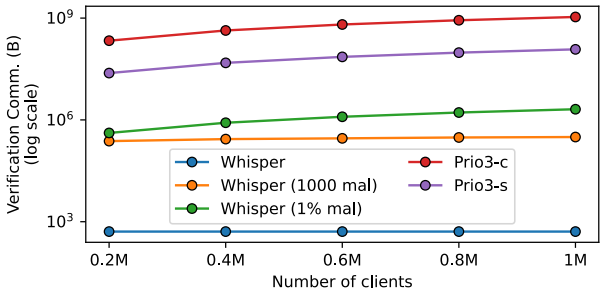


Figure 2.7: Verification communication per server with an increasing number of clients for Hist 1024.

borrows from Poplar's codebase [40]. 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 [4, 8, 64, 87]: 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.

General statistics

Baseline. We first compare with the state-of-the-art system Prio3 [48, 73]. 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 2.5 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 §2.4, 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 2.6 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 [62, 65], 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 2.7 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 2.5 and 2.6).

Heavy hitters

Baseline. We compare with Poplar [19], 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 [19]. We configure Poplar as in their evaluation. For

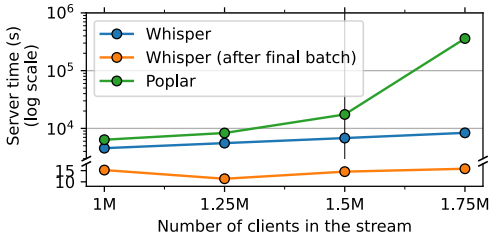


Figure 2.8: Server time to compute heavy hitters over a stream of client submissions for 0.1% threshold and 0.05% malicious clients. Poplar runs out of the main memory at the vertical line.

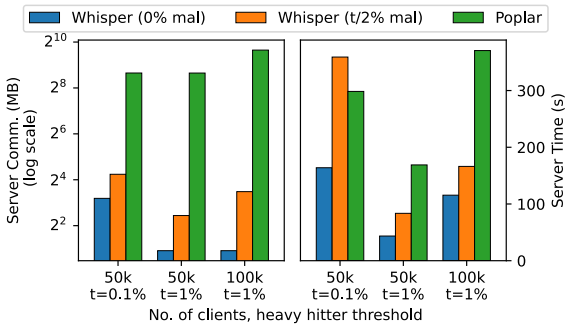


Figure 2.9: Server-to-server communication and time of each server to compute heavy hitters.

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 2.8 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 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 2.9) and server-to-server communication is lower by one to two orders of magnitude (Figure 2.9). This translates to up to 3.8× reduction in the dollar cost to run the servers based on Google Cloud's pricing model [62, 65]. 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.

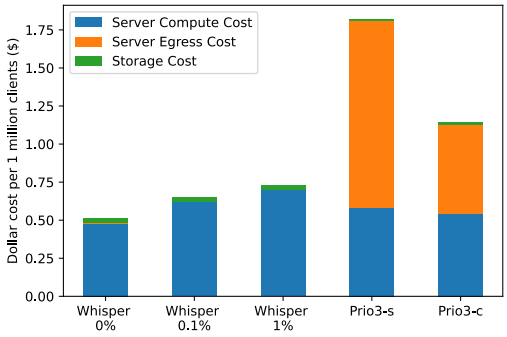


Figure 2.10: Dollar cost breakdown for ENPA aggregate statistics over 1 million clients

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 [64]. We simulate 1 million client uploads into two Google Cloud Storage Buckets, located in the same physical regions as their associated processing servers. We 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 [63] 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, according to our GCP price calculator estimations, we see between 1.5x-3.5x cost savings.

2.7 Related Work

Single-server model. There is a rich literature on systems for private collection of aggregate statistics via local differential privacy [5,12,13,29,85], 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 [5, 12, 13, 17, 29, 37, 51, 75, 85, 105]; adding such defenses can be expensive [14, 17, 53, 77, 83–85, 88].

Private heavy hitters. Mix-net [31,89] and other anonymous communication systems [42,55,56] 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 [10, 16, 74], cannot compute heavy hitters over a stream leading to large server-side storage [86] or both [19]. Star [44] 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 [86], 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 [80]. Moreover, Plasma and the two-server state-of-the-art Poplar [19] 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 [5, 12, 13, 25, 28, 29, 54, 94, 104] 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 [3, 19, 41, 86] 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 [19, 36, 41, 85], the noise is added directly to the aggregate [97]. This maintains higher accuracy compared to local differential privacy. Zhu et al. [37, 105] develop a trie-based heavy hitters protocol where subsampling the clients provides meaningful differential privacy without requiring additional noise. Prochlo [15] 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. [70] 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 [46] offer batch verification and succinct key sizes.

Chapter 3

Delegation Friendly SNARKs

3.1 Introduction

Zero Knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARKs) allow a prover to attest the knowledge of a witness that satisfies any given NP relation. The prover only needs to send a short proof to the verifier, and without any further communication, the verifier will be convinced that the prover has a satisfying witness, without learning anything about the prover's witness specifically. Their flexibility has led to a myriad of applications [22, 67, 103], including some from industry [2]. However, many zkSNARKs have extremely high compute and memory overhead for their proof generation protocols, limiting practicality.

A weak machine with a secret witness might want to *delegate* proof generation to a powerful server. *Public delegation* solutions parallelize well as we add more workers to the server, but leak the witness in its entirety to the server [81, 100, 101]. Since zkSNARKs are often computed over extremely sensitive information, there is a line of work for *private delegation* solutions [34,61,82,90]. In this setting, we leverage distributed trust to provide privacy: the server computing the zkSNARK proof is actually composed of multiple workers in different trust domains, and a client will upload secret shares of the witness onto each worker. The workers will then do some Multi Party Computation (MPC) over their shares in order to generate their proof. Though private delegation solutions can guarantee client privacy, they generally do not scale well – adding more workers can make it harder for attackers to compromise privacy, but performance gains are often limited.

Our Contribution

We first observe that certain subprotocols are well suited for public delegation, while others are more suited for private delegation. For example, the Fast Fourier Transform (FFT) can be efficiently computed under MPC, making it well suited for private delegation. However, all known algorithms for parallel computation of FFTs is bottlenecked by a single coordinator node, making it unsuited for public delegation. In contrast, zkSNARKs based on PLONK [60] usually involve a subprotocol called the grand product, which can be easily be parallelized among multiple workers without requiring any heavy coordination. However, expressing the grand product in an arithmetic circuit results in many multiplications, which makes it slow to compute under MPC. We construct DFS (Delegation Friendly zkSNARK) by only using subprotocols suited for both public and private delegation. Our construction starts with Spartan [96], a popular zkSNARK that achieves competitive performance in the single threaded case. Many of Spartan's underlying subprotocols are well suited to both public and private delegation. However, their memory checking subprotocol is not well suited for public delegation, so we replace it with the lookup protocol of [68]. We then formally describe how DFS naturally translates to the public delegation setting, where every worker is given full access to the witness. We additionally introduce novel MPC protocols tailored to our zkSNARK construction, and implement DFS in the private delegation setting. We evaluate DFS's performance in both the private and public delegation settings.

Related Work

Public delegation. DIZK [100] addresses the public delegation of zkSNARK proof generation. They specifically focus on distributing the workload of generating a Groth16 proof [66] across multiple workers, but their methods generalize to other types of proofs. In addition, zkBridge [101], a system that allows for effective communication across blockchains, introduces a specialized public distributed zkSNARK called deVirgo for batch verifying many digital signatures at once. Though deVirgo is specialized for data-parallel circuits, limiting its applications, it is able to scale gracefully as more compute nodes are added.

Both DIZK and deVirgo incur large communication costs between the provers. Pianist [81] solves this communication issue, and presents a performant solution for both data-parallel and non-data-parallel circuits. However, all of these solutions leak the entire witness to all workers.

Private Delegation. Collaborative zkSNARKs [90] divide up the witness into secret shares, and allow for proof generation under MPC. They provide building blocks for many different zkSNARKs and MPC protocols, but focus on Groth16 [66], Marlin [78], and Plonk [60].

EOS [34] considers a slightly weaker trust setting, where an honest, trusted, but computationally weak delegator gives secret shares of the witness to each worker, who then compute a zkSNARK proof under MPC. Each worker routes their communication through the trusted delegator, allowing for increased performance.

However, neither of these works see any performance improvements as they increase the number of workers. Adding additional workers can provide more privacy, but doesn't speed up proof generation.

Scalable Private Delegation. zkSAAS [61] is able to simultaneously preserve witness privacy among many provers while scaling with the number of workers. However, their protocol depends on the Fast Fourier Transform, which parallelizes poorly. A single server becomes a bottleneck, which limits their scaling – 128 workers leads to a mere 16x faster proving time over a single worker implementation.

A recent work from Liu et al. [82] successfully scales a different zkSNARK, Libra [102], in both the public and private delegation settings. However, they inherit weaknesses from their underlying zkSNARK – Libra works with layered arithmetic circuits, a more specific class of circuits than our general R1CS. Their proof size and verifier time is linear in the depth of the circuit, which may be large for certain computations. In addition, this work only scales for data-parallel circuits, while we achieve high parallelism for more general arithmetic circuits.

My Contribution

I worked on implementing and optimizing several subprotocols for the single worker case, including the lookup subprotocol and the polynomial commitment scheme. I helped develop and implement our single worker SNARK construction, and I formalized a large portion of the public delegated SNARK. Since I didn't work on the specialized MPC protocols, I elide most of their discussion in this report.

3.2 Preliminaries

Notation. We use λ to denote the security parameter. We use $\mathbb{F}$ to denote a finite field $\mathbb{Z}_q$ of prime order q. We will use bold lower-case letters like $\vec{x}$ for vectors, and denote by x_i the i-th component of $\vec{x}$ with x_0 the first entry. We also use colon notation to denote slices of vectors. For example $\vec{x}[1:]$ is every element of $\vec{x}$ except x_0. We use upper-case letters such as A to denote matrices.

Indexed relations. An *indexed relation* $\mathscr{R}$ is a set of triples (i, x, w) where i is the index, x is the instance, and w is the witness. An indexed *oracle* relation is an indexed relation where the index i and the instance x contain "implicit" inputs that are specified as oracles, i.e., the membership-checking algorithm for such a relation has only query access to these oracles. We adopt notation from [32] and use $[\![z]\!]$ to denote when the input z is provided as an oracle.

An important indexed relation in the SNARK literature is the R1CS relation, which we define next.

Definition 3.2.1 (R1CS indexed relation)**.** The indexed relation $\mathscr{R}_{\mathrm{R1CS}}$ is the set of all triples

$$(i, x, w) = \big((\mathbb{F}, n, m, A, B, C), x, w\big)$$

where $\mathbb{F}$ is a finite field, k, n, and m are natural numbers, $w \in \mathbb{F}^{m-|x|-1}$ is a vector over $\mathbb{F}$, A, B, C are $m \times m$ matrices over $\mathbb{F}$ with at most n nonzero entries, and $z := (x \;||\; 1 \;||\; w)$ is a vector in $\mathbb{F}^m$ such that $Az \circ Bz = Cz$. Here, $\circ$ denotes the Hadamard product. We assume that A, B, and C are square for simplicity.

Algebraic background. We will work over the n-dimensional Boolean hypercube $\{0,1\}^n$. The polynomial $\mathrm{eq}(X, Y) := \prod_{i=1}^{n}(X_iY_i + (1 - X_i)(1 - Y_i))$ checks that $X = Y$.

Sparse matrix encodings. Prior work [27, 33, 35] has shown how to represent (or arithmetize) a *square* matrix $M \in \mathbb{F}^{n \times n}$ with three univariate polynomials r_M, c_M, v_M, each of degree $\|M\|$.

Definition 3.2.2 (sparse matrix encodings)**.** Let $M \in \mathbb{F}^{m \times m}$ be a matrix with $\|n\|$ non-zero entries, and let $d = \log_2(\|n\|)$ and $s = \log_2(m)$. Then:

- the ***sparse matrix encoding*** of M is a triple of polynomials

$$\begin{pmatrix} \mathsf{v}_M : \{0,1\}^d \to \mathbb{F} \\ \mathsf{r}_M : \{0,1\}^d \to \mathbb{F} \\ \mathsf{c}_M : \{0,1\}^d \to \mathbb{F} \end{pmatrix}$$

  such that, for all $x \in \{0,1\}^d$, $\mathsf{v}_M(x) = M_{\mathsf{r}_M(x), \mathsf{c}_M(x)}$.
- the ***matrix-encoding polynomial*** of M is the bivariate polynomial

$$\tilde{M}(X, Y) = \sum_{z \in \{0,1\}^d} \mathsf{v}_M(z) \cdot \mathrm{eq}(\mathsf{r}_M(z), X) \cdot \mathrm{eq}(\mathsf{c}_M(z), Y)$$

  Here, the outputs of r_M and c_M are interpreted as elements of $\{0,1\}^s$.
  Notice that $\tilde{M}(x, y) = M[(x, y)]$.

We will omit the subscripts in r_M, c_M, and v_M when they are clear from context.

Background for zkSNARKs

A *succinct preprocessing non-interactive argument of knowledge* in the random oracle model (ROM) for an indexed relation $\mathscr{R}$ is a tuple of algorithms $\mathsf{ARG} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ satisfying completeness, knowledge soundness, succinctness, and zero knowledge. The indexer $\mathcal{I}$ preprocesses the NP index i into index-specific proving (ipk) and verification (ivk) keys. The prover $\mathcal{P}$, on input ipk, an instance x, and a witness w such that $(i, x, w) \in \mathscr{R}$, outputs a proof π which can be checked by the verifier $\mathcal{V}$ when given as input ivk and x.

In this work, we will focus on zkSNARKs constructed by following the approach of [33], which constructs a zkSNARK from a polynomial interactive oracle proof (PIOP) and a polynomial commitment (PC) scheme. We first describe these two primitives (§3.2), and then describe how to construct a zkSNARK from them (§3.3).

Polynomial commitments

A polynomial commitment scheme enables a sender to commit to a polynomial p and then later prove the correct evaluation of p at a desired point. Formally, it is a tuple of algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Trim}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Check})$ satisfying certain completeness, extractability, and hiding properties (see [33] for definitions of these). We are interested in particular in the Commit and Open algorithms:

- $\mathsf{PC.Commit}^{\rho}(\mathsf{ck}, p; \bar{p}) \to C$. On input the commitment key ck, a polynomial p over the field $\mathbb{F}$, PC.Commit outputs a commitment C to the polynomial p. The randomness $\bar{p}$ is used if the commitment C is hiding.
- $\mathsf{PC.Open}^{\rho}(\mathsf{ck}, C, p, z; \bar{p}) \to \pi_{\mathsf{PC}}$. On input the commitment key ck, a commitment C, the polynomial p committed inside C, an evaluation point $z \in \mathbb{F}$, and commitment randomness $\bar{p}$, PC.Open outputs an evaluation proof π_{PC}.

Polynomial interactive oracle proofs

A **polynomial interactive oracle proof** (PIOP) for an indexed relation $\mathscr{R}$ is an interactive protocol specified by a tuple $\mathsf{PIOP} = (\mathbb{F}, \mathsf{k}, \mathsf{s}, \mathbf{I}, \mathbf{P}, \mathbf{V})$ where $\mathbb{F}$ is a finite field, k is the number of rounds, $\mathsf{s}(j)$ is the number of prover polynomials in the j-th round, and $\mathbf{I}$, $\mathbf{P}$, $\mathbf{V}$ are algorithms described next.

In an offline phase, the indexer $\mathbf{I}$ preprocesses the NP index i into a set of *indexed polynomials* that are made available to the prover $\mathbf{P}$ (in full) and to the verifier $\mathbf{V}$ (as oracles).

During the online phase, $\mathbf{P}$ receives as input $(\mathbb{F}, i, x, w)$, while $\mathbf{V}$ receives as input $(\mathbb{F}, x)$, and has oracle access to the indexed polynomials. In each round $j \in [\mathsf{k}]$, $\mathbf{P}$ receives a message $\mu_j \in \mathbb{F}^*$ from $\mathbf{V}$ and replies with $\mathsf{s}(j)$ oracle polynomials $p_{j,1}, \ldots, p_{j,\mathsf{s}(j)} \in \mathbb{F}[X]$. $\mathbf{V}$ may query these polynomials (along with the indexed polynomials) any number of times. A query consists of a location $z \in \mathbb{F}$ for an oracle $p_{i,j}$, and its corresponding answer is $p_{i,j}(z) \in \mathbb{F}$. After the interaction, the verifier accepts or rejects. Every PIOP we consider in this paper is required to achieve perfect completeness, negligible knowledge soundness error, and zero knowledge. See [33] for details.

Constructing zkSNARKs from PIOPs and PC schemes

[33] constructs a zkSNARK from a PIOP and a PC scheme as follows. First, the argument indexer $\mathcal{I}$, on input i, invokes the PIOP indexer $\mathbf{I}$ to obtain the indexed polynomials, and commits to these using $\mathsf{PC.Commit}$. It then constructs ipk out of these polynomials and ck, and sets ivk to be the commitments.

The interactive argument prover $\mathcal{P}$ and verifier $\mathcal{V}$ respectively invoke the PIOP prover $\mathbf{P}$ and verifier $\mathbf{V}$. In each round, instead of directly sending the polynomial oracles output by $\mathbf{P}$, $\mathcal{P}$ instead commits to these polynomials via $\mathsf{PC.Commit}$, and sends the resulting commitments to $\mathcal{V}$, which invokes $\mathbf{V}$ to generate its next message. After the interaction, $\mathcal{V}$ invokes $\mathbf{V}$ to generate its queries to the committed polynomials. It sends these to $\mathcal{P}$, who replies with the desired evaluations along with an evaluation proof attesting to their correctness relative to the commitments. To obtain a zkSNARK, the Fiat–Shamir transform [58] is applied to this interactive argument.

PST13 polynomial commitment scheme

DFS uses this standard polynomial commitment scheme [92]. For efficiency, this construction requires a one-time, trusted setup, distributing the committer and verifier keys to the appropriate parties. This setup determines the maximum degree of the polynomials we can commit to. For the $\mathsf{PC.Commit}$ and $\mathsf{PC.Open}$ algorithms, we omit the hiding randomness $\bar{p}$ for simplicity; see [33] for details.

$\mathsf{PST.Setup}(1^\lambda, n) \to (\mathsf{ck}, \mathsf{rk})$:
1. Obtain $\langle \mathsf{group} \rangle = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. Sample random $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \leftarrow \mathbb{F}^n$.
3. Set $\boldsymbol{\Sigma} := [\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}$.
4. Set $\mathsf{ck} := (\boldsymbol{\Sigma}, \langle \mathsf{group} \rangle)$.
5. Set $\mathsf{rk} := ([\alpha_i \cdot H]_{i \in [n]}, \langle \mathsf{group} \rangle)$.
6. Output $(\mathsf{ck}, \mathsf{rk})$.

$\mathsf{PST.Commit}(\mathsf{ck}, p) \to C$:
1. Parse ck as $([\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}, G, H)$.
2. Output $C := \sum_{i \in \{0,1\}^n} p_i \cdot \mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G$.

$\mathsf{PST.Open}(\mathsf{ck}, p, z) \to \pi_{\mathsf{PC}}$:
Parse: $\mathsf{ck} = ([\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}, G, H)$.
1. Let $y := p(z)$
2. For each i in $[1, \ldots, n]$:
   a) Compute i-th witness polynomial $q_i(X)$ such that $p(X) - y = \sum_{i=1}^{n} q_i(X) \cdot (X_i - z_i)$
   b) Compute $\pi_i := q_i(\boldsymbol{\alpha}) \cdot G$.
3. Output evaluation proof $\pi_{\mathsf{PC}} := (\pi_1, \ldots, \pi_n)$.

$\mathsf{PST.Check}(\mathsf{rk}, C, z, v, \pi_{\mathsf{PC}}) \to \{0,1\}$:
Parse: $\mathsf{rk} = ([\alpha_i \cdot H]_{i\in[n]}, G, H)$ and $\pi_{\mathsf{PC}} = (\pi_1, \ldots, \pi_n)$.
1. Accept if $e(C - vG, H) = \sum_{i=1}^{n} e(\pi_i, (\alpha_i - z_i) \cdot H)$.

Common PIOPs

We now recall some common PIOPs that we will use in our construction of DFS. We omit their completeness, soundness, and zero-knowledge proofs as these can be found in prior work [32,98,99]

Sumcheck PIOP. Throughout this paper, we will be tasked with checking that an n-variate polynomial p sums to a claimed value σ over an n-dimensional Boolean hypercube $\{0,1\}^n$. This is formalized via the following relation:

Definition 3.2.3 (Sumcheck relation)**.** The relation $\mathscr{R}_{\mathrm{SC}}$ is the set of all triples $(i, x, w) = ((\mathbb{F}, n, d), (\llbracket p \rrbracket, \sigma), p)$ such that $\sum_{X\in\{0,1\}^n} p(X) = \sigma$.

The PIOP below illustrates a standard way of proving this relation.

PIOP for Sumcheck

For each i in $1, \ldots, n$:
1. If $i = 1$, $\mathbf{V}$ sets $\sigma_i := \sigma$; otherwise, it sets $\sigma_i := p_{i-1}(r_{i-1})$.
2. $\mathbf{P}$ computes $p_i(X_i) := \sum_{b_{i+1},\ldots,b_n\in\{0,1\}^{n-i}} p(r_1, \ldots, r_{i-1}, X_i, b_{i+1}, \ldots, b_n)$ and sends it to $\mathbf{V}$.
3. $\mathbf{V}$ checks that $p_i(0) + p_i(1) = \sigma_i$.
4. $\mathbf{V}$ samples a random point $r_i \in \mathbb{F}$ and sends it to $\mathbf{P}$.

Zerocheck PIOP. Throughout this paper, we will be tasked with checking that an n-variate polynomial p is zero at all points of an n-dimensional Boolean hypercube $\{0,1\}^n$. This is formalized via the following relation:

Definition 3.2.4 (Zerocheck relation)**.** The relation $\mathscr{R}_{\mathrm{ZC}}$ is the set of all triples $(i, x, w) = ((\mathbb{F}, n, d), \llbracket p \rrbracket, p)$ such that $p(X) = 0$ for all $X \in \{0,1\}^n$.

The PIOP below illustrates a standard way of proving this relation.

PIOP for Zerocheck

$\mathbf{P}$ has input p, while $\mathbf{V}$ has oracle access to p.
1. $\mathbf{V}$ samples a random point $r \in \mathbb{F}^n$ and sends it to $\mathbf{P}$.
2. $\mathbf{P}$ and $\mathbf{V}$ invoke the sumcheck PIOP for the claim "$\sum_{X\in\{0,1\}^n} p(X) \cdot \mathrm{eq}(X, r) = 0$".

Lookup PIOP. Another important building block is the lookup PIOP, where $\mathbf{P}$ has a *query* vector $\vec{q} \in \mathbb{F}^{2^n}$ and and a pre-shared *table* vector $\vec{t} \in \mathbb{F}^{2^m}$. The prover's goal is to assert that all elements of the query vector are contained in the table vector. In practice, the query and table vectors are represented as the evaluations of polynomials over the boolean hypercube.

This problem is formalized via the following relation:

Definition 3.2.5 (lookup)**.** The relation $\mathscr{R}_{\mathrm{LU}}$ is the set of all triples $(i, x, w) = ((\mathbb{F}, n, m, d), (\llbracket q \rrbracket, \llbracket t \rrbracket), (q, t))$ such that the sets $q(x)_{x\in\{0,1\}^n}$ is a subset of $\{t(x)\}_{x\in\{0,1\}^m}$.

The PIOP below illustrates a standard way of proving this relation that is adapted from [68].

PIOP for Lookup

P gets as input (q, t), while **V** gets as input $(\llbracket q \rrbracket, \llbracket t \rrbracket)$.

1. **P** receives a random challenge $r \in \mathbb{F}$ from **V**.
2. **P** computes polynomials $h_1(X), h_2(X)$ such that for each $x \in \{0,1\}^n$, $h_1(x) = (r + q(x))^{-1}$, and for each $x \in \{0,1\}^m$, $h_2(x) = (r + t(x))^{-1}$. That is, h_1 and h_2 are multilinear extensions of $(r + q(x))^{-1}$ and $(r + t(x))^{-1}$, respectively.
3. **P** sends $\llbracket h_1 \rrbracket$ and $\llbracket h_2 \rrbracket$ to **V**.
4. **P** evaluates $k := \sum_x h_1(x)$. Then, **P** and **V** invoke two Sumcheck PIOPs: one for the claim "$\sum_x h_1(x) = k$", and another for the claim "$\sum_x h_2(x) = k$".
5. **P** and **V** invoke a Zerocheck PIOP for the claim "$(r + q(X))h_1(X) - 1 = 0$".
6. **P** and **V** invoke a Zerocheck PIOP for the claim "$(r + t(X))h_2(X) - 1 = 0$".

This PIOP shows how to perform lookups over scalars, where each element of the query or table vectors is a single field element. If we want to check multi-set equality over tuples of field elements, we require a slight modification.

Definition 3.2.6 (batch lookup)**.** The relation $\mathscr{R}_{\mathrm{BLU}}$ is the set of all triples $(i, x, w) = ((\mathbb{F}, n, m, d), (\llbracket q_1 \rrbracket, \llbracket q_2 \rrbracket, \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket), (q_1, q_2, t_1, t_2))$ such that the set $\{(q_1(x), q_2(x))\}_{x\in\{0,1\}^n}$ is a subset of $\{t_1(x), t_2(x)\}_{x\in\{0,1\}^m}$.

PIOP for BatchLookup

P gets as input (q_1, q_2, t_1, t_2), while **V** gets as input $(\llbracket q_1 \rrbracket, \llbracket q_2 \rrbracket, \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket)$.

1. **P** receives a random challenge $r \in \mathbb{F}$ from **V**.
2. **P** computes the polynomial $q^*(x) := q_1(x) + rq_2(x)$ and the polynomial $t^*(x) := t_1(x) + rt_2(x)$.
3. **P** and **V** invoke a Lookup PIOP for the $q^*(x)$ and $t^*(x)$ polynomials. **V** can use r and $(\llbracket q_1 \rrbracket, \llbracket q_2 \rrbracket, \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket)$ in order to get an oracle over $q^*(x)$ and $t^*(x)$.

3.3 DFS: a delegation-friendly zkSNARK

We combine the common PIOPs of §3.2 to form a PIOP for R1CS satisfiability, inspired from that of Spartan [96].

High level overview

Before we start generating our proof, we first need to efficiently encode the matrices A, B, C and z. Each matrix $M \in \{A, B, C\}$ is expressed as r_M, c_M, and v_M, and z is expressed as the multilinear polynomial $\tilde{z}$. The verifier is given oracle access to all of these polynomials, as well as oracle access to the witness.

We start by defining the following polynomials.

$$\begin{pmatrix} \hat{A}(x) := \sum_{\vec{y}\in\{0,1\}^s} \tilde{A}(\vec{x},\vec{y})\tilde{z}(\vec{y}) \\ \hat{B}(x) := \sum_{\vec{y}\in\{0,1\}^s} \tilde{B}(\vec{x},\vec{y})\tilde{z}(\vec{y}) \\ \hat{C}(x) := \sum_{\vec{y}\in\{0,1\}^s} \tilde{C}(\vec{x},\vec{y})\tilde{z}(\vec{y}) \end{pmatrix}$$

Notice that Az, interpreted as a polynomial in evaluation form, is equal to $\hat{A}(x)$. This means that $Az \circ Bz = Cz$ only if $\tilde{F}(x) = \hat{A}(x) \cdot \hat{B}(x) - \hat{C}(x)$ is the zero polynomial. The prover and verifier will engage in a Zerocheck PIOP 2, which reduces to proving $e_x := \tilde{F}(\vec{r}_x)\mathrm{eq}(\vec{r}_x, r)$, where r is the zerocheck challenge, and r_x is the sumcheck challenge. The prover will provide claimed evaluations of $v_A := \hat{A}(\vec{r}_x), v_B := \hat{B}(\vec{r}_x), v_C := \hat{C}(\vec{r}_x)$, and the verifier will check if $(v_A \cdot v_B - v_C)\mathrm{eq}(\vec{r}_x, r) \stackrel{?}{=} e_x$, quickly evaluating $\mathrm{eq}(\vec{r}_x, r)$ locally.

Next, in order to verify the authenticity of v_A, v_B, and v_C, we perform a batched sumcheck. The prover receives random challenges r_A, r_B and r_C from the verifier, and we batch $\tilde{A}$, $\tilde{B}$, and $\tilde{C}$ together as $M_{\vec{r}_x}(\vec{y}) := r_A \cdot \tilde{A}(\vec{r}_x, \vec{y}) + r_B \cdot \tilde{B}(\vec{r}_x, \vec{y}) + r_C \cdot \tilde{C}(\vec{r}_x, \vec{y})$. Then, we perform a sumcheck for the following claim.

$$r_A v_A + r_B v_B + r_C v_C \stackrel{?}{=} \sum_{\vec{y}\in\{0,1\}^m} M_{\vec{r}_x}(\vec{y})\tilde{z}(\vec{y})$$

This, too, turns into a claimed evaluation $e_y \stackrel{?}{=} M_{\vec{r}_x}(\vec{r}_y)\tilde{z}(\vec{r}_y)$. In order for the verifier to efficiently query $\tilde{z}(\vec{r}_y)$, we have the prover send an oracle to witness extension $\tilde{w}$ at the start of the protocol. For simplicity, we assume that $\|w\| = \|x\| + 1$. If this is the case, then the verifier can easily evaluate

$$v_Z := (1 - \vec{r}_y[0])\tilde{w}(\vec{r}_y[1:]) + \vec{r}_y[0](\tilde{x}||1)(\vec{r}_y[1:])$$

At this point, the verifier needs a fast, succinct way to evaluate $\tilde{A}$, $\tilde{B}$, and $\tilde{C}$ at $(\vec{r}_x, \vec{r}_y)$. Recall that for all $M \in \{A, B, C\}$, $\tilde{M}(\vec{r}_x, \vec{r}_y) = \sum_i \mathsf{v}_M(i)\mathrm{eq}(\mathsf{r}_M(i), \vec{r}_x)\mathrm{eq}(\mathsf{c}_M(i), \vec{r}_y)$. We start with a sumcheck claim over i, which results in evaluating $\mathsf{v}_M(r_z)$, $\mathrm{eq}(\mathsf{r}_M(r_z), \vec{r}_x)$, and $\mathrm{eq}(\mathsf{c}_M(r_z), \vec{r}_y)$ at a random point r_z. The verifier can easily check v_M using its oracle. To evaluate $\mathrm{eq}_{row}(x) := \mathrm{eq}(\mathsf{r}_M(x), \vec{r}_x)$and $\mathrm{eq}_{col}(x) := \mathrm{eq}(\mathsf{c}_M(r_z), \vec{r}_y)$, the prover can send over oracles to the verifier.

Now, the prover needs to show that their oracle to $\text{eq}_{row}(x)$ is well formed. Spartan's solution, memory checking, is unsuitable here. At a high level, Spartan's memory checker would verify eq_{row} by constructing it entry by entry using read and write operations. Since these operations need to work over the entire matrix, memory checking is hard to distribute over multiple workers [21]. Instead, we use our batch lookup PIOP from earlier. If we show that

$$\{(\mathsf{r}_M(x), \text{eq}(\mathsf{r}_M(x), \vec{r}_x)) : x \in \{0,1\}^d\} \subset \{(i, \text{eq}(i, \vec{r}_x)) : i \in \{0,1\}^s\}$$

we can be convinced that the provided oracle to $\text{eq}_{row}(x)$ was valid. The same process applies for $\text{eq}_{col}(x)$. We now present a more formal construction of these ideas.

Formal Construction

PIOP for R1CS:

Indexer I: on input $(\mathbb{F}, n, m, A, B, C)$, proceeds as follows:

1. For each $M \in \{A, B, C\}$:
   a) Derive r_M, c_M, and v_M from M. Output these polynomials.

P gets as input $\tilde{z}$, as well as $\tilde{A}$, $\tilde{B}$, and $\tilde{C}$, while **V** gets as input x and $[\![\mathsf{r}_M]\!], [\![\mathsf{c}_M]\!], [\![\mathsf{v}_M]\!]$ for M in {A B C}.

1. **P** computes $\hat{M}(x) := \sum_{\vec{y}\in\{0,1\}^s} \tilde{M}(\vec{x}, \vec{y})\tilde{z}(\vec{y})$, for $M \in A, B, C$. Then, **P** computes $\tilde{F}(\vec{x}) = \hat{A}(\vec{x}) \cdot \hat{B}(\vec{x}) - \hat{C}(\vec{x})$.
2. **P** and **V** invoke the Zerocheck PIOP (PIOP 2) on the polynomial $\tilde{F}$. This leads to an evaluation claim of the form $\tilde{F}(\vec{r}_x) * \text{eq}(r, \vec{r}_x) = e_x$ for a zerocheck challenge r and random point $\vec{r}_x \in \{0,1\}^s$.
3. To answer this claim, **P** computes $v_M := \hat{M}(\vec{r}_x)$ for each $M \in \{A, B, C\}$, and sends v_A, v_B, v_C to **V**.
4. **V** asserts that $e_x \stackrel{?}{=} (v_A \cdot v_B - v_C)$.

5. **V** randomly samples $r_A, r_B, r_C \in \mathbb{F}^s$, and sends them to **P**.
6. **P** computes $M_{\vec{r}_x}(\vec{y}) := (r_A \cdot \tilde{A}(\vec{r}_x, \vec{y}) + r_B \cdot \tilde{B}(\vec{r}_x, \vec{y}) + r_C \cdot \tilde{C}(\vec{r}_x, \vec{y}))\tilde{z}(\vec{y})$.
7. **P** and **V** engage in a Sumcheck PIOP for the claim "$\sum_{\vec{y}\in\{0,1\}^s} M_{\vec{r}_x}(\vec{y}) = r_A v_A + r_B v_B + r_C v_C$".
8. This leads to an evaluation claim of the form $e_y \stackrel{?}{=} M_{\vec{r}_x}(\vec{r}_y)$, where $\vec{r}_y \in \{0,1\}^s$ is a random evaluation point.
9. **V** evaluates $v_Z := (1 - \vec{r}_y[0])\tilde{w}(\vec{r}_y[1:]) + \vec{r}_y[0](\tilde{x}||1)(\vec{r}_y[1:])$. Then, the verifier asserts that $e_y \stackrel{?}{=} (r_A \cdot \tilde{A}(\vec{r}_x\vec{r}_y) + r_B \cdot \tilde{B}(\vec{r}_x, \vec{r}_y) + r_C \cdot \tilde{C}(\vec{r}_x, \vec{r}_y)) \cdot v_Z$

10. For each $M \in \{A, B, C\}$:
    a) **P** and **V** invoke a Sumcheck PIOP for the claim $\sum_i \mathsf{v}(i)\text{eq}(\mathsf{r}_M(i), \vec{r}_x)\text{eq}(\mathsf{c}_M(i), \vec{r}_y) = \tilde{M}(\vec{r}_x, \vec{r}_y)$, resulting in a random challenge $\vec{r}_z$ and claimed evaluation e_z.
    b) **P** sends oracles for $\text{eq}_{row}(x) := \text{eq}(\mathsf{r}_M(x), \vec{r}_x)$ and $\text{eq}_{col}(x) := \text{eq}(\mathsf{c}_M(x), \vec{r}_x)$. **V** uses these oracles to assert $\mathsf{v}_M(\vec{r}_z)\text{eq}_{row}(\vec{r}_z)\text{eq}_{col}(\vec{r}_z) \stackrel{?}{=} e_z$
    c) **P** and **V** invoke the batched lookup PIOP (PIOP 4) where $q_1(x) := \mathsf{r}_M(x)$, $q_2(x) := \text{eq}_{row}(x)$, t_1

is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(x) := \mathrm{eq}(x, \vec{r}_x)$.

d) **P** and **V** invoke the batched lookup PIOP for $q_1(x) := \mathsf{c}_M(x)$, $q_2(x) := \mathrm{eq}_{col}(x)$, t_1 is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(x) := \mathrm{eq}(x, \vec{r}_y)$.

Discussion

The verifier needs only to participate in a constant number of sumchecks over s and d-variate polynomials. This results in a verifier time complexity and proof size of $O(\log(n) + \log(m))$.

The prover work, alongside the aforementioned sumchecks, additionally requires some polynomial operations. All of these take time linear in either the number of constraints or the number of variables, leading to a prover time complexity of $O(n + m)$.

3.4 Public delegation

DFS naturally lends itself to the *public delegation* setting, where N_w trusted workers collaborate to generate a proof with the help of a central coordinator $\mathbf{P}^*$. Workers can all see the full witness in this setting. We denote each worker as $\mathbf{P}^{(j)}$, for $j \in [N_w]$, and the set of all workers as $[\mathbf{P}]$. For convenience, we denote $n_w := \log(N_w)$.

Distributed PIOPs

We require distributed versions of the PIOPs defined in Section 3.2. In general, whenever we have an n-variate polynomial $p(x)$, we can split it into N_w parts $p_j(x)$, each of which is $n - n_w$-variate. If $p(x)$ is in evaluation form, this can easily be done by giving each worker 2^{n-n_w} consecutive evaluations. Each worker essentially has the original n-variate polynomial with the first n_w variables fixed. $\mathbf{P}^{(0)}$ has the first n_w variables set to $(0, 0, \cdots, 0)$, $\mathbf{P}^{(1)}$ has the first n_w variables set to $(0, 0, \cdots, 1)$, etc.

Distributed Sumcheck and Zerocheck

For the public delegation setting, we can make a simple adjustment to the non distributed sumcheck PIOP. For the first $n - n_w$ verifier challenges, each worker will locally perform their own sumcheck, and the coordinator will be responsible for the last n_w challenges.

PIOP for Publicly Delegated Sumcheck

1. For each i in $1, \ldots, n - n_w$:
   a) If $i = 1$, $\mathbf{V}$ sets $\sigma_i := \sigma$; otherwise, it sets $\sigma_i := p_{i-1}(r_{i-1})$. $\mathbf{V}$ broadcasts σ_i to all provers $\mathbf{P}^{(j)}$ for $j \in [n_w]$.
   b) All workers $\mathbf{P}^{(j)}$ compute $p_i^{(j)}(X_i) := \sum_{b_{i+1},\ldots,b_{n-n_w} \in \{0,1\}^{n-n_w-i}} p^{(j)}(r_1, \ldots, r_{i-1}, X_i, b_{i+1}, \ldots, b_n)$ and sends it to $\mathbf{V}$.
   c) $\mathbf{V}$ checks that $\sum_j p_i^{(j)}(0) + p_i^{(j)}(1) = \sigma_i$.
   d) $\mathbf{V}$ samples a random point $r_i \in \mathbb{F}$ and broadcasts it to all $\mathbf{P}^{(j)}$.
2. Each worker computes $y^{(j)} := p^{(j)}(r_1, r_2, \ldots, r_{n-n_w})$ and sends it to the coordinator, $\mathbf{P}^*$.
3. $\mathbf{P}^*$ interpolates the n_w-variate polynomial $p^*(X)$ from the evaluations $y^{(j)}$.
4. $\mathbf{P}^*$ and $\mathbf{V}$ engage in the non-distributed sumcheck protocol for the instance $(\llbracket p^* \rrbracket, \sigma_{n-n_w})$ and the witness p^*, where σ_{n-n_w} is final σ_i from the previous loop.

Publically Delegated Zerocheck. Using this building block, we can create a distributed version of the PIOP 2 in a straightforward fashion. We call this the Distributed Zerocheck PIOP.

PIOP for Publicly Delegated Zerocheck

$\mathbf{P}^{(j)}$ has input p_j, while $\mathbf{V}$ has oracle access to p.

1. $\mathbf{V}$ samples a random point $r \in \mathbb{F}^n$ and sends it to $[\mathbf{P}]$.
2. $\mathbf{P}^{(j)}$ computes the partial evaluation of $\mathrm{eq}(x, r)$ corresponding to their id j. The j'th worker will

take evaluations from $j \cdot (n - n_w)$ to $(j+1) \cdot (n - n_w)$ to form their own $\mathrm{eq}_j(x, r)$
3. Each worker defines $q_j(x) := p_j(x) \cdot \mathrm{eq}_j(x, r)$ and uses this part to do a Distributed Sumcheck for the claim $\sum q(x) = 0$.

Discussion. For the first $n - n_w$ rounds of the publicly delegated sumcheck, the coordinator sends one random challenge to all workers, and each worker sends a constant number of evaluations to the coordinator. This results in a total of $O(\log_2(n - n_w))$ round trip communication between the workers for the whole protocol

Distributed Lookup

We also require a distributed version of the lookup PIOP 3. This occurs in the same setting, where each worker $\mathbf{P}^{(j)}$ has a $n - n_w$-variate and $m - n_w$-variate chunk of the polynomials q and t respectively.

PIOP for Publicly Delegated Lookup

Each worker $\mathbf{P}^{(j)}$ gets as input $q^{(j)}(X), t^{(j)}(X)$, while $\mathbf{V}$ gets as input $([\![q]\!], [\![t]\!])$.

1. Each worker $\mathbf{P}^{(j)}$ receives the same random challenge $r \in \mathbb{F}$ from $\mathbf{V}$.
2. $\mathbf{P}^{(j)}$ computes polynomials $h_1^{(j)}(X), h_2(X)$ such that for each $x \in \{0,1\}^{n-n_w}$, $h_1^{(j)}(x) = (r + q^{(j)}(x))^{-1}$, and for each $x \in \{0,1\}^{m-n_w}$, $h_2^{(j)}(x) = (r + t^{(j)}(x))^{-1}$.
3. $\mathbf{P}^{(j)}$ evaluates $k_j := \sum h_1^{(j)}(X)$, and sends it to $\mathbf{P}^*$, who sums up every $k := \sum k_j$.
   Then, $[\mathbf{P}]$ and $\mathbf{V}$ invoke two Distributed Sumcheck PIOPs: one for the claim "$\sum_x h_1(x) = k$", and another for the claim "$\sum_x h_2(x) = k$".
4. $[\mathbf{P}]$ and $\mathbf{V}$ invoke a Distributed Zerocheck PIOP for the claim "$(r + q(X))h_1(X) - 1 = 0$".
5. $[\mathbf{P}]$ and $\mathbf{V}$ invoke a Distributed Zerocheck PIOP for the claim "$(r + p_2(X))h_2(X) - 1 = 0$".

Discussion. This can naturally be extended to the batched multiset equality PIOP from PIOP 4. We refer to this as the distributed batch lookup PIOP.

Distributed PST13 polynomial commitment scheme

We additionally require a distributed polynomial commitment scheme, where each worker gets a small part of the secret polynomial. The workers need to work together to jointly compute the commitment and openings of their polynomial. Similar to the distributed sumcheck protocol, each worker will receive a smaller committer key, as well a $n - n_w$-variate chunk of the secret polynomial,

$\mathsf{DistPST.Setup}(1^\lambda, n) \to ([\mathsf{ck}^{(j)}], \mathsf{ck}^*, \mathsf{rk})$:
1. Obtain $\langle \mathsf{group} \rangle = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. Sample random $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \leftarrow \mathbb{F}^n$. Denote $\boldsymbol{\alpha}^* = (\alpha_1, \ldots \alpha_{n_w})$, and $\boldsymbol{\alpha}^w = (\alpha_{n_w+1}, \ldots, \alpha_n)$
3. Set $\boldsymbol{\Sigma_j} := [\mathrm{eq}(\boldsymbol{\alpha}, i \;||\; j) \cdot G]_{i \in \{0,1\}^{n-n_w}}$, where j is the worker index in binary form.
4. Set $\boldsymbol{\Sigma^*} := [\mathrm{eq}(\boldsymbol{\alpha}^*, i) \cdot G]_{i \in \{0,1\}^{n_w}}$
5. Set $\mathsf{ck}^{(j)} := (\boldsymbol{\Sigma_j}, \langle \mathsf{group} \rangle)$.
6. Set $\mathsf{ck}^* := (\boldsymbol{\Sigma^*}, \langle \mathsf{group} \rangle)$.
7. Set $\mathsf{rk} := ([\alpha_i \cdot H]_{i \in [n]}, \langle \mathsf{group} \rangle)$.
8. Output $([\mathsf{ck}^{(j)}], \mathsf{ck}^*, \mathsf{rk})$.

$\mathsf{DistPST.Commit}([\mathsf{ck}^{(j)}], p^{(j)}) \to C$:
1. Let $C^{(j)} = \mathsf{PST.Commit}(\mathsf{ck}^{(j)}, p^{(j)})$.
2. Output $C = \sum_{j \in [N_w]} C^{(j)}$

$\mathsf{DistPST.Open}([\mathsf{ck}^{(j)}], \mathsf{ck}^*, [p^{(j)}], z) \to \pi_{\mathsf{PC}}$:
1. Let $z^* := z[: n_w]$, and $z_w := z[n_w + 1 : n]$.
2. Initialize $\pi^w_{\mathsf{PC}} \in \mathbb{F}^{n-n_w} := (0, \cdots, 0)$.
3. For all $j \in [N_w]$, parse $\mathsf{ck}^{(j)} = ([\mathrm{eq}(\boldsymbol{\alpha}, j \;||\; i) \cdot G]_{i \in \{0,1\}^{n-n_w}}, G, H)$.
   a) Let $y^{(j)} := p^{(j)}(z_w)$
   b) For each i in $[1, \ldots, n - n_w]$:
      i. Compute i-th witness polynomial $q_i^{(j)}(X)$ such that $p^{(j)}(X) - y^{(j)} = \sum_{i=1}^{n-n_w} q_i^{(j)}(X) \cdot (X_i - z_w[i])$.
      ii. Compute $\pi_i^{(j)} := q_i^{(j)}(\boldsymbol{\alpha}) \cdot G$.
      iii. Update $\pi^w_{\mathsf{PC}}[i] = \pi^w_{\mathsf{PC}}[i] + \pi_i^{(j)}$
   c) Output $r^{(j)} := p^{(j)}(z_w)$, which is a byproduct of the $\pi^{(j)}_{\mathsf{PC}}$ computation.
4. Interpolate the n_w variate polynomial p^* from the evaluations $(r^{(0)}, r^{(1)}, \cdots, r^{(N_w)})$.
5. Let $\pi^*_{\mathsf{PC}} := \mathsf{PST.Open}(\mathsf{ck}^*, p^*, z^*)$.
6. Output $\pi_{\mathsf{PC}} := \pi^w_{\mathsf{PC}} \;||\; \pi^*_{\mathsf{PC}}$.

$\mathsf{DistPST.Check}(\mathsf{rk}, C, z, v, \pi_{\mathsf{PC}}) \to \{0, 1\}$:
This procedure is the same as the non distributed version.

Discussion. During DistPST.Commit, each worker only needs to send a single group element commitment back to the coordinator. During DistPST.Open, each worker will send $O(n - n_w)$ items back to the coordinator: each $r^{(j)}$ and the $n - n_w$ updates to π^w_{PC}.

Distributed PIOP for R1CS

Using these building blocks, we can modify our PIOP for R1CS satisfiability PIOP 5 for the public delegation setting.

PIOP for Publicly Delegated R1CS:

Indexer I: The distributed indexer mostly works the same as the non-distributed indexer. However, we split r_M, c_M, and v_M into N_w parts for each $Min\{A, B, C\}$. As before, we denote teh j'th worker's part of a certain polynomial p as $p^{(j)}$. each $\mathsf{r}_M, \mathsf{c}_M, \mathsf{v}_M$ is a $n - n_w$-variate polynomial.

$\mathbf{P}^{(j)}$ gets as input $\tilde{z}_j$, as well as the $\mathsf{r}_M^{(j)}, \mathsf{c}_M^{(j)}, \mathsf{v}_M^{(j)}$ polynomials as described in the indexer. In addition, we precompute Az, Bz, Cz, and distribute the results to each worker as the $s - n_w$-variate polynomials $\hat{A}^{(j)}(x), \hat{B}^{(j)}(x), \hat{C}^{(j)}(x)$. As before, $\mathbf{V}$ gets as input x and $[\![\mathsf{r}_M]\!], [\![\mathsf{c}_M]\!], [\![\mathsf{v}_M]\!]$ for M in $\{A, B, C\}$.

1. $\mathbf{P}^{(j)}$ computes $\tilde{F}^{(j)}(\vec{x}) = \hat{A}^{(j)}(\vec{x}) \cdot \hat{B}^{(j)}(\vec{x}) - \hat{C}^{(j)}(\vec{x})$.
2. $[\mathbf{P}]$ and $\mathbf{V}$ invoke the Distributed Zerocheck PIOP (PIOP 7) on the polynomial $\tilde{F}$. This leads to an evaluation claim of the form $\tilde{F}(\vec{r}_x)\mathrm{eq}(\vec{r}_x, r) = e_x$ for a random point $\vec{r}_x \in \{0,1\}^s$, as well as the zerocheck challenge r.
3. For each $M \in \{A, B, C\}$
   a) $\mathbf{P}^{(j)}$ computes $v_M^{(j)} := \hat{M}^{(j)}(\vec{r}_x)$, and sends $v_M^{(j)}$ to $\mathbf{P}^*$.
   b) $\mathbf{P}^*$ sums $v_M := \sum_j v_M^{(j)}$, and sends v_M to $\mathbf{V}$.
4. $\mathbf{V}$ asserts that $e_x \stackrel{?}{=} (v_A \cdot v_B - v_C)\mathrm{eq}(\vec{r}_x, r)$. Again, $\mathbf{V}$ evaluates $\mathrm{eq}(\vec{r}_x, r)$ locally.

5. $\mathbf{V}$ randomly samples $r_A, r_B, r_C \in \mathbb{F}^m$, and broadcasts them to $[\mathbf{P}]$.
6. Each $\mathbf{P}^{(j)}$ individually computes $M_{\vec{r}_x}^{(j)}(\vec{y}) := (r_A \cdot \tilde{A}^{(j)}(\vec{r}_x, \vec{y}) + r_B \cdot \tilde{B}^{(j)}(\vec{r}_x, \vec{y}) + r_C \cdot \tilde{C}^{(j)}(\vec{r}_x, \vec{y}))\tilde{z}_j(\vec{y})$.
   - Recall that $\tilde{M}(\vec{r}_x\vec{y}) := \sum_{z \in \{0,1\}^d} \mathsf{v}_M(z) \cdot \mathrm{eq}(\mathsf{r}_M(z), \vec{r}_x) \cdot \mathrm{eq}(\mathsf{c}_M(z), \vec{y})$. For $\mathbf{P}^{(j)}$ to compute $\tilde{M}^{(j)}(\vec{r}_x, \vec{y})$, they need only to compute $\mathrm{eq}(\mathsf{r}_M^{(j)}(z), \vec{r}_x)$, which is fairly cheap.
7. $[\mathbf{P}]$ and $\mathbf{V}$ engage in a Distributed Sumcheck PIOP for the claim "$\sum_{\vec{y} \in \{0,1\}^s} M_{\vec{r}_x}(\vec{y}) = r_A v_A + r_B v_B + r_C v_C$". This leads to an evaluation claim of the form $M_{\vec{r}_x}(\vec{y})$, where $\vec{r}_y \in \{0,1\}^s$ is a random evaluation point.
8. $\mathbf{V}$ evaluates $v_Z := (1 - \vec{r}_y[0])\tilde{w}(\vec{r}_y[1:]) + \vec{r}_y[0](\tilde{x}||1)(\vec{r}_y[1:])$. Then, the verifier asserts that $e_y \stackrel{?}{=} (r_A \cdot \tilde{A}(\vec{r}_x, \vec{r}_y) + r_B \cdot \tilde{B}(\vec{r}_x, \vec{r}_y) + r_C \cdot \tilde{C}(\vec{r}_x, \vec{r}_y)) \cdot v_Z$

9. For each $M \in \{A, B, C\}$:
   a) $[\mathbf{P}]$ and $\mathbf{V}$ invoke a Distributed Sumcheck PIOP for the claim $\sum \mathsf{v}(i)\mathrm{eq}(\mathsf{r}_M(i), \vec{r}_x)\mathrm{eq}(\mathsf{c}_M(i), \vec{r}_y) = \tilde{M}(\vec{r}_x, \vec{r}_y)$, resulting in a random challenge $\vec{r}_z$ and claimed evaluation e_z.
   b) $[\mathbf{P}]$ use the distributed polynomial commitment scheme §3.4 sends oracles for $\mathrm{eq}_{row}(x) := \mathrm{eq}(\mathsf{r}_M(x), \vec{r}_x)$ and $\mathrm{eq}_{col}(x) := \mathrm{eq}(\mathsf{c}_M(x), \vec{r}_x)$.
   $\mathbf{V}$ uses these oracles to assert $\mathsf{v}_M(\vec{r}_z)\mathrm{eq}_{row}(\vec{r}_z)\mathrm{eq}_{col}(\vec{r}_z) \stackrel{?}{=} e_z$
   c) $[\mathbf{P}]$ and $\mathbf{V}$ invoke the Distributed batched lookup PIOP (PIOP 4) where $q_1(x) := \mathsf{r}_M(x)$, $q_2(x) := \mathrm{eq}_{row}(x)$, t_1 is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(x) := \mathrm{eq}(x, \vec{r}_x)$.
   d) $[\mathbf{P}]$ and $\mathbf{V}$ invoke the Distributed batched lookup PIOP for $q_1(x) := \mathsf{c}_M(x)$, $q_2(x) := \mathrm{eq}_{col}(x)$, t_1 is the same as in the previous step, and $t_2(x) := \mathrm{eq}(x, \vec{r}_y)$.

Single prover time	Distributed prover time per party	Distributed communication cost	Verifier time	Proof size
$O(m+n)$	$O(\frac{(m+n)}{N_w})$	$O(\log(m)+\log(n))$	$O(\log(m)+\log(n))$	$O(\log(m)+\log(n))$

Table 3.1: Asymptotics for the single prover and public delegated prover case. m is the number of R1CS constraints, n is the total number of inputs for the R1CS, and N_w is the number of workers.

Discussion

After every verifier challenge, each $\mathbf{P}^{(j)}$ does some local computation, and then participates in one of the distributed PIOPs from 3.4. Each of these distributed PIOPs only incurs $O(\log(m))$ or $O(\log(n))$ communication cost, so each worker's communication complexity is only $O(\log(m) + \log(n))$. In addition, each worker gets exactly the same amount of work in our construction.

The coordinator's work mostly lies in networking with the workers. The only special compute the coordinator is responsible for is operations over n_w-variate polynomials, as in the last phase of DistPST.Open and the distributed sumcheck, so the coordinator compute is merely $O(N_w)$. We summarize these asymptotics in §3.4.

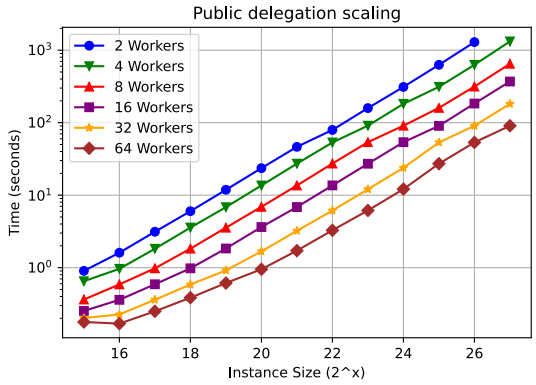


Figure 3.1: DFS's proof generation time as we increase the number of workers.

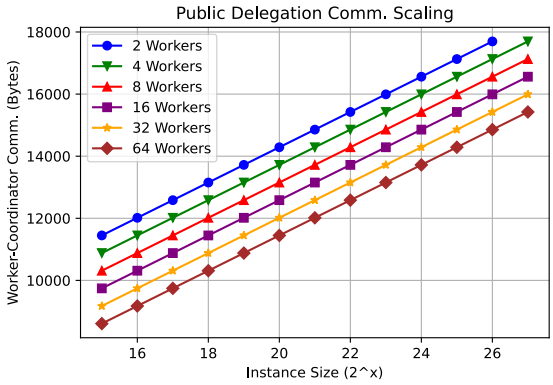


Figure 3.2: DFS's worker-coordinator communication in the public delegation setting.

3.5 Implementation and Performance Evaluation

Implementation

We implemented DFS in Rust, on top of the `arkworks` library [9]. We evaluated our system on a university on-prem cluster of machines, where each node has a Intel Xeon Scalable Cascade Lake 6248 CPU and 16GB memory. Nodes are connected with a 100GBPS link — this link is significantly faster than the evaluations of prior work, but our logarithmic communication complexity makes this difference negligible in practice. Though each node's CPU technically has 20 cores, we implement our solution using only 8 threads per worker. This is due to some technical limitations in the `arkworks` library.

Evaluation and comparison

We seek to answer the following questions in our evaluation:

- How does DFS's latency scale as we increase the size of the R1CS instance and increase the number of workers?
- How does DFS's communication cost scale as we add more compute nodes?

Scaling workers for public delegation

We evaluate DFS in the public delegation setting (§3.5). Proof generation time scales gracefully with the number of workers – doubling the number of workers roughly halves the proving time. Our experiment with 2 workers and 2^{27} constraints ran out of memory, so it failed to run.

Communication

As shown in §3.5, communication also scales gracefully as we increase the number of workers. Since adding more workers results in each worker being responsible for a smaller portion of the total workload, their individual communication with the coordinator decreases. However, due to increasing communication overhead, total communication across all workers slightly increases.

Bibliography

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

[2] Zcash, 2020. https://z.cash/.

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

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

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

[6] Apple. Managed device attestation for apple devices. https://support.apple.com/guide/deployment/managed-device-attestation-dep28afbde6a/web.

[7] Apple. Mitigate fraud with AppAttest and DeviceCheck. *WWDC21*, 2021.

[8] 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.

[9] arkworks contributors. arkworks zksnark ecosystem, 2022.

[10] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *CCS*, 2022.

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

[12] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. In *NIPS*, 2017.

[13] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, 2015.

[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*, 2023.

[15] 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*, 2017.

[16] Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In *CCS*, 2021.

[17] 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 *CCS*, 2017.

[18] 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 (3)*, 2019.

[19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy*, 2021.

[20] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Arithmetic sketching. In *CRYPTO (1)*, 2023.

[21] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic snarks for diverse environments. Cryptology ePrint Archive, Paper 2022/420, 2022. `https://eprint.iacr.org/2022/420`.

[22] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P '20, pages 947–964, 2020.

[23] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.

[24] Anne Broadbent and Alain Tapp. Information-theoretic security without an honest majority. In *ASIACRYPT*, 2007.

[25] Mark Bun, Jelani Nelson, and Uri Stemmer. Heavy hitters and the structure of local privacy. *ACM Trans. Algorithms*, 15, 2019.

[26] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA:privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security*, 2010.

[27] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodriguez. Lunar: a toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions. In *Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '21, 2021.

[28] Karan N. Chadha, Junye Chen, John C. Duchi, Vitaly Feldman, Hanieh Hashemi, Omid Javidbakht, Audra McMillan, and Kunal Talwar. Differentially private heavy hitter detection using federated analytics. *CoRR*, abs/2307.11749, 2023.

[29] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. Differentially private continual monitoring of heavy hitters from distributed streams. In *Privacy Enhancing Technologies*, 2012.

[30] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312, 2004.

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

[32] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '23, pages 499–530, 2023.

[33] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '20, 2020.

[34] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zksnark provers. In *Proceedings of the 32nd USENIX Security Symposium*, USENIX Security '23, pages 6453–6469, 2023.

[35] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '20, 2020.

[36] Seung Geol Choi, Dana Dachman-Soled, Mukul Kulkarni, and Arkady Yerukhimovich. Differentially-private multi-party sketching for large-scale statistics. *Proc. Priv. Enhancing Technol.*, 2020.

[37] Graham Cormode and Akash Bharadwaj. Sample-and-threshold differential privacy: Histograms and applications. In *AISTATS*, 2022.

[38] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 2008.

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

[40] Henry Corrigan-Gibbs. heavyhitters. `https://github.com/henrycg/heavyhitters`.

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

[42] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, 2015.

[43] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella Béguelin. Smart meter aggregation via secret-sharing. In *SEGS@CCS*, 2013.

[44] Alex Davidson, Peter Snyder, E. B. Quirk, Joseph Genereux, Benjamin Livshits, and Hamed Haddadi. STAR: secret sharing for private threshold aggregation reporting. In *CCS*, 2022.

[45] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Proc. Priv. Enhancing Technol.*, 2023.

[46] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In *EUROCRYPT (1)*, 2022.

[47] Divvi Up. `https://divviup.org/`.

[48] Divvi Up. `https://github.com/divviup/libprio-rs`.

[49] Robert Dorfman. The Detection of Defective Members of Large Populations. *The Annals of Mathematical Statistics*, 14(4), 1943.

[50] Dingzhu Du, Frank K Hwang, and Frank Hwang. *Combinatorial group testing and its applications*, volume 12. World Scientific, 2000.

[51] 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*, 2010.

[52] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, 2008.

[53] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *CCS*, 2014.

[54] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[55] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *NDSS*, 2022.

[56] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security Symposium*, 2021.

[57] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. TVA: A multi-party computation system for secure and expressive time series analytics. In *USENIX Security Symposium*, 2023.

[58] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[59] 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, 2020.

[60] Ariel Gabizon. Improved prover efficiency and SRS size in a sonic-like system. 2019/601, 2019.

[61] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zksaas: Zero-knowledge snarks as a service. In *Proceedings of the 32nd USENIX Security Symposium*, USENIX Security '23, pages 4427–4444, 2023.

[62] Google. All networking pricing. `https://cloud.google.com/vpc/network-pricing`.

[63] Google. exposure-notifications-android. `https://github.com/google/exposure-notifications-android/`.

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

[65] Google. VM instance pricing. `https://cloud.google.com/compute/vm-instance-pricing`.

[66] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, 2016.

[67] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. Cryptology ePrint Archive, Paper 2021/1022, 2021. `https://eprint.iacr.org/2021/1022`.

[68] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. 2022/1530, 2022.

[69] Shai Halevi, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. Additive randomized encodings and their applications. In *CRYPTO (1)*, 2023.

[70] Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. Your reputation's safe with me: Framing-free distributed zero-knowledge proofs. *IACR Cryptol. ePrint Arch.*, page 1523, 2022.

[71] Thomas Humphries, Rasoul Akhavan Mahdavi, Shannon Veitch, and Florian Kerschbaum. Selective MPC: distributed computation of differentially private key-value statistics. In *CCS*, 2022.

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

[73] IETF. Verifiable distributed aggregation functions draft-irtf-cfrg-vdaf-07. `https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/`.

[74] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Poster: Vogue: Faster computation of private heavy hitters. In *CCS*, 2022.

[75] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *Privacy Enhancing Technologies*, 2012.

[76] Alan F. Karr, Xiaodong Lin, Ashish P. Sanil, and Jerome P. Reiter. Regression on distributed databases via secure multi-party computation. In *DG.O*, 2004.

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

[78] O(1) Labs. Marlin.

[79] Kasper Green Larsen, Jelani Nelson, Huy L. Nguyen, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *FOCS*, 2016.

[80] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. The deployment dilemma: Merits & challenges of deploying mpc. `https://mpc.cs.berkeley.edu/blog/deployment-dilemma`.

[81] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. Cryptology ePrint Archive, Paper 2023/1271, 2023. `https://eprint.iacr.org/2023/1271`.

[82] Xuanming Liu, Zhelei Zhou, Yinghao Wang, Bingsheng Zhang, and Xiaohu Yang. Scalable collaborative zk-snark: Fully distributed proof generation and malicious security. Cryptology ePrint Archive, Paper 2024/143, 2024. `https://eprint.iacr.org/2024/143`.

[83] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. Rofl: Robustness of secure federated learning. In *SP*, 2023.

[84] Elizabeth Margolin, Karan Newatia, Tao Luo, Edo Roth, and Andreas Haeberlen. Arboretum: A planner for large-scale federated analytics with differential privacy. In *SOSP*, 2023.

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

[86] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. Plasma: Private, lightweight aggregated statistics against malicious adversaries with full security. Cryptology ePrint Archive, Paper 2023/080.

[87] 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/`.

[88] Moni Naor, Benny Pinkas, and Eyal Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In *CCS*, 2019.

[89] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS*, 2001.

[90] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security '22, pages 4291–4308, 2022.

[91] Rasmus Pagh. Compressed matrix multiplication. *ACM Trans. Comput. Theory*, 5, 2013.

[92] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 222–242, 2013.

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

[94] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS*, 2016.

[95] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. Cryptology ePrint Archive, Paper 2022/878.

[96] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Proceedings of the 40th Annual International Cryptology Conference*, CRYPTO '20, pages 704–737, 2020.

[97] Thomas Steinke. Multi-central differential privacy. *CoRR*, abs/2009.05401, 2020.

[98] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 71–89, 2013.

[99] Justin Thaler. Proofs, arguments, and zero-knowledge. *Found. Trends Priv. Secur.*, 4(2-4):117–660, 2022.

[100] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *Proceedings of the 27th USENIX Security Symposium*, USENIX Security '18, pages 675–692, 2018.

[101] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical, 2022.

[102] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Proceedings of the 39th Annual International Cryptology Conference*, CRYPTO '19, pages 733–764, 2019.

[103] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2039–2053, 2020.

[104] Mingxun Zhou, Tianhao Wang, T.-H. Hubert Chan, Giulia Fanti, and Elaine Shi. Locally differentially private sparse vector aggregation. In *SP*, 2022.

[105] Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated heavy hitters discovery with differential privacy. In *AISTATS*, 2020.