Lightweight Techniques for Private Heavy Hitters

Dan Boneh, *Stanford*
Elette Boyle, *IDC Herzliya*
Henry Corrigan-Gibbs, *EPFL and MIT CSAIL*
Niv Gilboa, *Ben-Gurion University*
Yuval Ishai, *Technion*

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00048

Abstract. This paper presents a new protocol for solving the *private heavy-hitters problem*. In this problem, there are many clients and a small set of data-collection servers. Each client holds a private bitstring. The servers want to recover the set of all popular strings, without learning anything else about any client's string. A web-browser vendor, for instance, can use our protocol to figure out which homepages are popular, without learning any user's homepage. We also consider the simpler *private subset-histogram problem*, in which the servers want to count how many clients hold strings in a particular set without revealing this set to the clients.

Our protocols use two data-collection servers and, in a protocol run, each client send sends only a single message to the servers. Our protocols protect client privacy against arbitrary misbehavior by one of the servers and our approach requires no public-key cryptography (except for secure channels), nor general-purpose multiparty computation. Instead, we rely on *incremental distributed point functions*, a new cryptographic tool that allows a client to succinctly secret-share the labels on the nodes of an exponentially large binary tree, provided that the tree has a single non-zero path. Along the way, we develop new general tools for providing malicious security in applications of distributed point functions.

A limitation of our heavy-hitters protocol is that it reveals to the servers slightly more information than the set of popular strings itself. We precisely define and quantify this leakage and explain how to ameliorate its effects. In an experimental evaluation with two servers on opposite sides of the U.S., the servers can find the 200 most popular strings among a set of 400,000 client-held 256-bit strings in 54 minutes. Our protocols are highly parallelizable. We estimate that with 20 physical machines per logical server, our protocols could compute heavy hitters over ten million clients in just over one hour of computation.

1 Introduction

To improve their products, manufacturers of hardware devices and software applications collect information about how their products perform in practice. For example, when your web browser crashes today, it prompts you to send an error report to the vendor with the URL that triggered the crash. For the browser-vendor, it is important to know which URLs are responsible for the majority of crashes. But since these crash reports contain the URLs that you (the user) have been visiting, sending these reports leaks information about your browsing history to the vendor. It takes just one subsequent data breach or one malicious insider to expose these reports—and the information they contain about your browsing history—to the world.

This data-collection task is an instance of the *private heavy-hitters problem*. In this problem, there are many clients and a small set of data-collection servers. Each client holds a string (e.g., a URL that caused a browser crash). For some threshold $t \in \mathbb{N}$, the servers want to recover every string that more than t clients hold. In this and other applications, each client's string comes from a large universe (the set of all URLs), so any solution that requires enumerating over the set of all possible strings is infeasible.

This problem comes up in an array of private data-collection applications: a cellphone vendor wants to learn which mobile apps consume the most minutes of user attention per day, without learning how much each person uses each app, an electric-car company wants to learn on which roads its cars most often run low on battery, without learning which car was where, and so on.

In this paper, we solve this private heavy-hitters problem using a new suite of lightweight cryptographic techniques. These tools are relatively simple to implement, are concretely efficient, unlike methods based on general-purpose multiparty computation [32, 47], and outperform existing approaches based on secure aggregation [18, 40]. We expect the cryptographic tools developed in this work to be useful in other contexts.

We work in the setting in which clients communicate with two non-colluding data-collection servers. The system protects *client privacy* as long as one of the two servers is honest (the other may deviate arbitrarily from the protocol and may collude with an unbounded number of malicious clients). For example, the maintainer of an app store could run one server and the app developer could run the other. The system protects *correctness* against any number of malicious clients. That is, the worst a malicious client can do to disrupt the protocol's execution is to lie about its own input string.

Our protocols require no public-key cryptographic operations, apart from those needed to establish secret channels between the parties. In terms of communication, if each client holds an n-bit string and we want to achieve λ-bit security, each client sends a single message, of roughly λn bits, to the servers (ignoring low-order terms). Since our schemes require each client to send only a single message to the servers, our schemes naturally tolerate unreliable clients: each client needs to stay online only long enough to send its single message to the servers. In a deployment with C clients, the servers communicate λnC bits with each other (again, ignoring low-order terms). In terms of computation, the client invokes a length-doubling pseudorandom generator, such as AES in counter mode, $O(n)$ times. When searching for strings that more than a $\tau \in (0, 1]$ fraction of clients hold, the servers

DOI 10.1109/SP40001.2021.00048

perform $\approx nC/\tau$ evaluations of a length-doubling pseudorandom generator.

To evaluate our heavy-hitter protocols in practice, we implement the end-to-end system and evaluate it on Amazon EC2 machines on opposite sides of the U.S. In this cross-country configuration, we consider a set of 400,000 clients, each holding a 256-bit string (long enough to hold a 40-character domain name). We configure the two servers to compute the set of heavy hitters held by more than 0.1% of these clients. The protocol between the two servers takes 54 minutes in total and requires under 70 KB of communication per user. With this parameter setting, our approach concretely requires over 100× less communication (between the servers) and server-side computation compared to approaches based on existing cryptographic tools.

Our techniques. Our first step to solving the private heavy-hitters problem is to study an independently useful simpler problem of computing *private subset histograms*. In this problem, each client holds an n-bit string, as before. Now, the servers have a small set S of strings (unknown to the clients) and, for each string $\sigma \in S$, the servers want to know how many clients hold string σ, without learning anything else about any client's string. Our starting point is a simple protocol for this problem from prior work [11], in which each client sends each server a single message. This protocol relies on the cryptographic tool of distributed point functions [10, 11, 31]. (A distributed point function is essentially a compressed secret-sharing of a function that has a single non-zero output.) The prior protocol [11] offers a partial defense against malicious clients at the expense of compromising the privacy of clients against a malicious server.

Our first technical contribution is to modify this protocol to simultaneously protect correctness against malicious clients and achieve privacy against a malicious server. To do so, we develop a new lightweight malicious-secure protocol that the two servers can run to check that they hold additive secret shares of a vector that is zero everywhere except with a one in a single position. Prior approaches either required additional non-colluding servers [19], did not provide malicious security [11], had relatively large client-to-server communication (as in Prio [18]), or required additional rounds of interaction between the clients and servers [28]. Applying our new building-block immediately improves the efficiency of existing privacy-preserving systems for advertising [45] and messaging [19, 28, 46].

Perhaps even more important, prior protocols [11] do not defend against a subtle "double-voting" attack. In this attack, a malicious client can cast tentative votes for a set S' of two or more strings. The servers only catch the cheating client if $|S' \cap S| \geq 2$, where S is the set of strings that the server whose counts the To prevent this kind of attack, we leverage a refined type of distributed point functions that we term *extractable distributed point functions* ("extractable DPFs"). Roughly speaking, with an extractable DPF it is possible to extract from the actions of a malicious client an honest strategy that would achieve a similar effect. We show that a variant of the distributed-point-function construction of prior work [11] is extractable in this sense when we model the underlying PRG as a random oracle.

Next, we use our protocol for private subset histograms to construct a protocol for the t-heavy hitters problem. Our approach follows that of prior work which uses subset-histograms protocols, in the settings streaming and local-differential privacy, to identify heavy hitters [3, 16, 17, 48].

In the t-heavy hitters problem, each client i holds a string $\alpha_i \in \{0,1\}^n$ and the servers want to learn the set of all strings that more than t clients hold, for some parameter $t \in \mathbb{N}$. Our idea is to have the client and servers run our private subset-histogram protocol n times. After the ℓth execution of the subset-histogram protocol, the servers learn a set $S_\ell \subseteq \{0,1\}^\ell$ that contains the ℓ-bit prefix of every t-heavy hitter. After n executions, the servers learn the set S_n of all t-heavy hitter strings.

In more detail, the clients, for their part, participate in n executions of the subset-histogram protocol. In the ℓth execution, for $\ell = 1, \ldots, n$, a client holding a string $\alpha \in \{0,1\}^n$ participates in the protocol using the prefix $\alpha|_\ell \in \{0,1\}^\ell$ as its input to the protocol, where $\alpha|_\ell$ is the ℓ-bit prefix of α. These executions all run in parallel, so each client in fact only sends a single message to the servers.

The servers participate in the first execution of the subset-histogram protocol using the set of two prefixes $S_1 = \{0,1\}$, and learn the histogram for this set S_1 (i.e., the number of client strings that begin with a '0' and the number of client strings that being with a '1'). They prune from S_1 all the prefixes that occur fewer than t times. Let $T_1 \subseteq S_1$ be the remaining set of prefixes. The servers then append a '0' and a '1' to every string in T_1 to obtain the set $S_2 = T_1 \times \{0,1\}$. In the second execution of the subset-histogram protocol, the servers learn the histogram for the set S_2. Again, they prune from S_2 all the elements that occur fewer than t times. Let $T_2 \subseteq S_2$ be the remaining set of prefixes. They compute $S_3 = T_2 \times \{0,1\}$, and learn the histogram for S_3. They prune S_3 and continue this way until finally after n executions of the subset-histogram protocol, they obtain the set T_n of all t-heavy hitters. At every step in this protocol, the size of the set S_ℓ is at most twice the size of the final answer T_n.

The straightforward implementation of the above scheme requires each client to communicate $\Omega(n^2)$ bits to each server, where n is the length of each client's private string. This is because each client participates in n instances of the private-subset-histogram protocol, and each protocol run requires the client to send a size-$\Omega(n)$ distributed-point-function key to the servers. Since $n \approx 256$ in our applications, the quadratic per-client communication cost is substantial.

To reduce this cost, we introduce *incremental distributed point functions* ("incremental DPFs"), a new cryptographic primitive that reduces the client-to-server communication from quadratic in the client's string length n to linear in n. Conceptually, this primitive gives the client a way to succinctly secret-share the weights on a tree that has a single path of non-zero weight in an incremental fashion.

Limitations. The main downside of our heavy-hitters protocols is that they reveal some additional—though modest and precisely quantified—information to the data-collection servers about the distribution of client-held strings, in addition to the set of heavy hitters itself. In particular, even when an arbitrary number of malicious clients collude with a malicious server, this leakage depends only on the *multiset* of strings held by the honest clients, without revealing any association between clients and strings in

this multiset. Moreover, the amount of partial information leaked about this multiset is comparable to the output length, and only scales logarithmically with the number of clients C when the servers search for strings that a constant fraction of clients hold. See the precise definition of the leakage in Section 2.3.

To protect client privacy against even this modest leakage, we can configure the system to provide ϵ-differential privacy [25], in addition to its native MPC-style security properties. The differential-privacy guarantee then ensures that the system will never reveal "too much" about any client's string, even accounting for the leakage. To achieve ϵ-differential privacy with C clients, our protocols introduce additive $O(1/\epsilon)$ error, compared with the larger $\Omega(\sqrt{C}/\epsilon)$ error inherent to protocols based on randomized response [3, 4, 12, 27, 29]. (Our protocols provide additional privacy benefits that cannot be obtained via randomized response, such as the ability to securely compute on the secret-shared histogram.)

An additional limitation is that our techniques require two non-colluding servers and they do not efficiently scale to the setting of k servers, tolerating $k-1$ malicious servers. Overcoming this limitation would either require constructing better multi-party distributed point functions [11] or using a completely different approach.

Contributions. The main contributions of this work are:

1. a malicious-secure protocol for private heavy hitters in the two-server setting,
2. a malicious-secure protocol for private subset histograms in the two-server setting,
3. the definition and construction of incremental and extractable distributed point functions,
4. a new malicious-secure protocol for checking that a set of parties hold shares of a vector of weight at most one, and
5. implementation and evaluation of our heavy-hitters protocol.

2 Problem statement

We work in a setting in which there are two data-collection *servers*. Our system provides privacy as long as at least one of these two servers executes the protocol faithfully. (The other server may maliciously deviate from the protocol.) There is some number C of participating *clients*. Each client i, for $i \in \{1, \ldots, C\}$, holds a private input string $\alpha_i \in \{0,1\}^n$. The goal of the system is to allow the servers to compute some useful aggregate statistic over the private client-held strings $(\alpha_1, \ldots, \alpha_C)$, while leaking as little as possible to the servers about any individual client's string.

Notation. Throughout the paper we use $\mathbb{F}$ to denote a prime field and $\mathbb{G}$ a finite Abelian group, we use $[n]$ to denote the set of integers $\{1, \ldots, n\}$, and $\mathbb{N}$ to denote the natural numbers. We let $\mathbf{1}\{P\}$ be the function that returns 1 when the predicate P is true, and returns 0 otherwise. We denote assignments as $x \leftarrow 4$ and, for a finite set S, the notation $x \xleftarrow{R} S$ indicates a uniform random draw from S. For strings a and b, $a\|b$ denotes their concatenation.

2.1 Private-aggregation tasks

In this setting, there are two tasks we consider.

Task I: Subset histogram. In this task, the servers hold a set $S \subseteq \{0,1\}^n$ of strings. For each string $\sigma \in S$, the servers want to learn the number of clients who hold the string σ. In some of our applications, both the clients and servers know the set S (i.e., the set is public). In other applications, the servers choose the set S and may keep it secret.

As a concrete application, a web-browser vendor may want to use subset histograms to privately measure the incidence of *homepage hijacking* [41]. A user's homepage has been "hijacked" if malware changes the user's homepage browser setting without her consent. In this application, the browser vendor has a set S of URLs it suspects are benefiting from homepage hijacking. The vendor wants to know, for each URL $u \in S$, how many clients have URL u as their homepage. For this application, it is important that the browser vendor hide the set S of suspect websites from the clients—both to avoid legal liability and to prevent these sites from taking evasive action.

In this application then, each client i's string α_i would be a representation of her homepage URL. The servers' set $S = \{\sigma_1, \sigma_2, \ldots\}$ would be the set of suspect URLs. And then the output of the task would tell the browser vendor how many clients use each of these suspect URLs as a homepage, without revealing to the servers which client has which homepage.

Task II: Heavy hitters. In this task, the servers want to identify which strings are "popular" among the clients. More precisely, for an integer $t \in \mathbb{N}$, we say that a string σ is a *t-heavy hitter* if σ appears in the list $(\alpha_1, \ldots, \alpha_C)$ more than t times. The t-heavy hitters task is for the servers to find all such strings. Note that, unlike the previous subset histogram task, here there is no *a priori* set of candidate heavy hitters.

As an illustrative application, consider a web browser vendor who wants to learn which URLs most crash the browser for more than 1000 clients. Each client i's string α_i is a representation of the last URL its browser loaded before crashing. The t-heavy hitters in the list $(\alpha_1, \ldots, \alpha_C)$, for $t = 1000$, reveal to the servers which URLs crashed the browser for more than 1000 clients. The servers learn nothing about which client visited which URL, nor do they learn anything about URLs that caused browser crashes for fewer than 1000 clients.

2.2 Communication pattern

While we primarily focus on the two tasks mentioned above — subset histogram and heavy hitters — the protocols we design can be described more generally as protocols for privately computing an aggregate statistic $agg = f(\alpha_1, \ldots, \alpha_C)$ over the data $\alpha_1, \ldots, \alpha_C \in \{0,1\}^n$ of the C clients, where the function f is known to the servers but possibly not to the clients.

Because we do not allow communication between clients, and minimal communication between the clients and the servers, the communication pattern for the private aggregation protocol should be as follows:

- *Setup:* In an optional setup phase, the servers generate public parameters, which they send to all clients.

- *Upload:* The clients proceed in an arbitrary order, where each participating client sends a single message to Server 0 and a single message to Server 1. Alternatively, the client can send a single message to Server 0 that includes an encryption of its second message, which Server 0 then routes to Server 1. We allow no other interaction with or between the clients.
- *Aggregate:* Servers 0 and 1 execute a protocol among themselves, and output the resulting aggregate statistic agg. This step may involve multiple rounds of server-to-server interaction.

All of the protocols that we consider in this paper obey the above communication pattern.

2.3 Security properties

Our private-aggregation protocols are designed to provide the following security guarantees. In the full version of this work [8], we provide formal security definitions.

Completeness: If all clients and all servers honestly follow the protocol, then the servers correctly learn $agg = f(\alpha_1, \ldots, \alpha_C)$.

Robustness to malicious clients: Informally, a malicious client cannot bias the computed aggregate statistic agg beyond its ability to choose its input $\alpha \in \{0,1\}^n$ arbitrarily. The same should hold for a coalition of malicious clients working together, where each can cast at most a single vote. Whether a malicious client's vote is counted or not may depend on the set S (for subset histogram) or on other client inputs (for heavy hitters).

Privacy against a malicious server: Informally, if one of the servers is malicious, and the other is honest, the malicious server should learn nothing about the clients' data beyond the aggregate statistic agg. Furthermore, even if a malicious adversary corrupts both a server and a subset of the clients, the adversary should learn no more than it could have learned by choosing the inputs of malicious clients and observing the output agg.

Our private subset-histogram protocol in Section 4 indeed meets this ideal goal, revealing to the adversary only the subset histogram of the participating honest clients. A malicious server can choose to "disqualify" honest clients independently of their input, so that their input does not count towards the output. (As a simple example, the server could pretend to not receive any message from a certain client.) The differentially private mechanism in Section 7 protects honest clients from being singled out via this attack. Alternatively, if too many clients are disqualified, the honest server can abort the computation.

Our most efficient heavy-hitters protocols in Section 5 reveal to a malicious adversary, who corrupts one server and a subset of the clients, a small amount of information about the honest client data beyond the list of t-heavy hitters. We capture this using a *leakage function* $L : (\{0,1\}^n)^C \to \{0,1\}^\ell$ that describes the extra information the adversary obtains. A malicious server should learn nothing about the client data beyond the agg and $L(\alpha_1, \ldots, \alpha_C)$. While we defer the full specification of the leakage function L to the full version of this work [8], we note here two important features of this function: first, L is *symmetric* in the sense that it only depends on the *multiset* of strings that the non-disqualified honest clients hold. In particular, the leakage reveals no association between clients and strings in this multiset. Second, the output length of L is comparable to that of agg, and only scales logarithmically with the number of clients C when $\tau = t/C$ is fixed. Thus, our protocol leaks typically much less than a shuffling-based approach that reveals the entire multiset. In particular, it does not often expose rare inputs, which are often the most sensitive.

Remark (Non threat: Correctness against malicious servers). If one of the servers maliciously deviates from the protocol, we do not guarantee that the other (honest) server will recover the correct value of the aggregate statistic. Prior private-aggregation systems offer a similarly relaxed correctness guarantee [2, 13, 14, 18, 22, 40, 45]. In practice, we typically run these protocol with two organizations that gain no advantage by corrupting the system's output. (In contrast, the organizations do potentially stand to benefit by learning the client's private data.) So, protecting correctness is less crucial than protecting client privacy. Protecting correctness in the presence of malicious servers would be a useful extension that we leave for future work.

2.4 Alternative approaches

We discuss a few alternative ways to solve these problems.

Mix-net. If the servers want to compute the multiset of *all* client-held strings (i.e., the threshold $t = 1$), the participants can just use a two-server mix-net [15]. That is, each client onion encrypts her string to the two servers, who each shuffle and decrypt the batch of strings. Using verifiable shuffles [42] prevents misbehavior by the servers. In the special case of $t = 1$ and with C clients, this alternative has computation cost $O(C)$ (hiding polynomial factors in the security parameter), while our protocols would have cost $O(C^2)$. However, without additional rounds of interaction between the clients and servers, the mix-net-based approach does not generalize to searching for t-heavy hitters with $t > 1$, where all non t-heavy hitters remain hidden. Our approach does.

Generic MPC + ORAM. Another alternative solution uses general-purpose malicious-secure two-party computation for RAM programs [30, 34, 37, 39]. Each client sends each server an additive secret-sharing of its input string. The servers then run a malicious-secure multiparty computation of a RAM program that takes as input C strings (one from each client) and computes the heavy hitters. This approach could have asymptotically optimal computational complexity $\widetilde{O}(C + t)$, for heavy-hitters threshold t. At the same time, multiparty computation of RAM programs—even without malicious security—is extremely expensive in concrete terms [23], as it requires implementing an oblivious RAM [33] client in a multiparty computation. There may be more sophisticated ways to, for example, efficiently implement a streaming algorithm for heavy hitters [17] in a multiparty computation. We expect that such techniques will be substantially more complicated to implement and will be concretely more expensive.

Counting data structures + secure aggregation. The count-min sketch [17] is a data structure used for finding approximate heavy hitters in the context of streaming algorithms. Melis et

al. [40] demonstrate that it is possible to use secure-aggregation techniques to allow each client to anonymously insert its input string into the data structure. When the set of candidate heavy hitters is unknown, as in our setting, it is possible to use a set of n such counting data structures (where each client holds an n-bit string) to recover the heavy hitters. The drawbacks of this approach are: (1) the concrete complexity is worse than our schemes since each client must send a large data-structure update message to each server (see Section 8), (2) the additional leakage is substantially larger and more difficult to quantify than in our protocol, and (3) these techniques only give approximate answers, where our techniques compute the heavy hitters exactly.

Local differential privacy. A beautiful line of work has considered protocols for computing heavy hitters in the *local model* of differential privacy, often using sophisticated variants of randomized response [3, 4, 12, 44, 48]. The advantage of these protocols is that they require only a single data-collection server. In contrast, our protocols and others based on multiparty computation require at least two non-colluding servers. The downside of these protocols is that they leak a non-negligible amount of information about each client's private string to the server. As we describe in Section 2.3, the leakage in our schemes depends only on the multiset of private client strings. Thus our protocols give incomparably stronger privacy guarantees and, as we discuss in Section 7, can also achieve differential privacy. In addition, when configured to provide differential privacy our protocols introduce less noise than those based on local differential privacy. (Since we have two non-colluding servers, the noise grows essentially as it would in the central model of differential privacy [25].)

3 Background

This section summarizes the existing techniques for private aggregation that we build on in this work.

A long line of work [2, 13, 22, 26, 35, 36, 40, 43, 45] has constructed private-aggregation schemes in the client/server model in which security holds as long as the adversary cannot control all servers. To demonstrate how these techniques work, consider the task of computing subset histograms (Task I of Section 2.1). Each client i holds a private string $\alpha_i \in \{0,1\}^n$ and the servers hold a set $S = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ of strings. For each $\sigma \in S$, the servers want to know how many clients hold the string σ.

Distributed point functions (DPFs). We can use *distributed point functions* [10, 11, 31] to accomplish this task in a privacy-preserving way. A distributed point function is, at a high level, a technique for secret-sharing a vector of 2^n elements in which only a single element is non-zero. The important property of distributed point functions is that each share has only size $O(n)$, whereas a naïve secret sharing would have share size 2^n.

More formally, a DPF scheme, parameterized by a finite field $\mathbb{F}$, consists of two routines:

- $\mathsf{Gen}(\alpha, \beta) \to (k_0, k_1)$. Given a string $\alpha \in \{0,1\}^n$ and value $\beta \in \mathbb{F}$, output two DPF keys representing secret shares of a dimension-2^n vector that has value $\beta \in \mathbb{F}$ only at the α-th position and is zero everywhere else.

Protocol 1: Private subset histograms. There are two servers and C clients. Each client i, for $i \in [C]$ holds a string $\alpha_i \in \{0,1\}^n$. The servers hold a set $S \subseteq \{0,1\}^n$ of strings. For each string $\sigma \in S$, the servers want to learn the number of clients who hold σ. The protocol uses a prime field $\mathbb{F}$ with $|\mathbb{F}| > C$.

The protocol is as follows:

1. Each client $i \in \{1, \ldots, C\}$, on input string $\alpha_i \in \{0,1\}^n$ prepares a pair of DPF keys as $(k_{i0}, k_{i1}) \leftarrow \mathsf{Gen}(\alpha_i, 1)$. The client sends k_{i0} to server 0 and k_{i1} to server 1.
2. For each string $\sigma_j \in S$, each server $b \in \{0,1\}$ computes the sum of its DPF keys evaluated at the string σ_j:

$$\mathsf{val}_{jb} \leftarrow \sum_{i=1}^{C} \mathsf{Eval}(k_{ib}, \sigma_j) \quad \in \mathbb{F}.$$

   Each server $\beta \in \{0,1\}$ then publishes the values

$$(\mathsf{val}_{1b}, \ldots, \mathsf{val}_{|S|b}) \quad \in \mathbb{F}^{|S|}.$$

3. Finally, for each string $\sigma_j \in S$, each server can conclude that the number of clients who hold string σ_j is $\mathsf{val}_{j0} + \mathsf{val}_{j1} \in \mathbb{F}$.

- $\mathsf{Eval}(k, x) \to \mathbb{F}$. Given a DPF key k and index $x \in \{0,1\}^n$, output the value of the secret-shared vector at the position indexed by the string x.

The DPF correctness property states that, for all strings $\alpha \in \{0,1\}^n$ output values $\beta \in \mathbb{F}$, keys $(k_0, k_1) \leftarrow \mathsf{Gen}(\alpha, \beta)$, and strings $x \in \{0,1\}^n$, it holds that

$$\mathsf{Eval}(k_0, x) + \mathsf{Eval}(k_1, x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases},$$

where the addition is computed in the finite field $\mathbb{F}$. Informally, the DPF security property states that an adversary that learns either k_0 or k_1 (but not both) learns no information about the special point α or its value β.

The latest DPF constructions [11], on a domain of size 2^n, have keys of length roughly λn bits, when instantiated with a length-doubling PRG that uses λ-bit keys.

A simple protocol for private subset histograms. Given DPFs, we can solve the subset-histogram problem using the following simple protocol, which we illustrate in Protocol 1. At a high level, each client i uses DPFs to create a secret sharing of a vector of dimension 2^n. This vector is zero everywhere except that it has "1" at the position indexed by client i's input string $\alpha_i \in \{0,1\}^n$. To learn how many clients hold a particular string σ, the servers can compute, for each client i, the shares of the σ-th value in the ith client's secret-shared vector. By publishing the sum of these shares, the servers learn exactly how many clients held string σ.

Correctness holds since

$$\begin{aligned}\mathsf{val}_{j0} + \mathsf{val}_{j1} &= \sum_{i=1}^{C} \mathsf{Eval}(k_{i0}, \sigma_j) + \sum_{i=1}^{C} \mathsf{Eval}(k_{i1}, \sigma_j) \\ &= \sum_{i=1}^{C} \big(\mathsf{Eval}(k_{i0}, \sigma_j) + \mathsf{Eval}(k_{1i}, \sigma_j)\big) \\ &= \sum_{i=1}^{C} \mathbf{1}\{\alpha_i = \sigma_j\},\end{aligned}$$

which is exactly the number of clients who hold string σ_j.

As long as one of the two servers is honest, a fully malicious adversary controlling the other server and any number of clients learns nothing about the honest clients' inputs, apart from what the subset histogram itself leaks.

In the following sections, we show how to extend this simple scheme to protect against corruption attacks by malicious clients (Section 4) and support computing heavy hitters (Section 5 and 6). In Section 7, we demonstrate that it is possible to achieve user-level differential privacy with these methods as well. Finally, in Section 8 we provide an experimental evaluation of the efficiency of the heavy-hitters protocol.

4 Privacy-preserving subset histograms via malicious-secure sketching

In this section, we show how to modify the simple scheme of Section 3 to protect against corruption attacks by malicious clients.

In the scheme of Section 3, if even *one* of the participating clients is malicious, it can completely corrupt the histogram that the servers recover. In particular, in Step 1 of the protocol above, a malicious client can send malformed DPF keys to the servers. A client who mounts this attack can prevent the servers from recovering any output (i.e., the servers get only pseudorandom garbage) or can manipulate the statistics (i.e., the client can arbitrarily influence the histogram the servers recover).

For example, if the servers are using this private-subset-histogram scheme to measure the incidence of homepage hijacking (cf. Section 2.1), a single malicious client could manipulate the output histogram to make it look as if no homepage hijacking was taking place.

4.1 Prior work: Sketching for malicious clients

Prior work [7, 11, 19, 28] has presented techniques to harden the simple scheme of Section 3 against misbehavior by malicious clients. These approaches use similar methods: before the servers accept the pair of DPF keys from the client, the servers check that the DPF keys are "well formed." That is, the two servers check that the DPF keys submitted by each client expand to shares of a vector that is zero everywhere and one at a single position.

More specifically, given a pair of client-submitted DPF keys (k_0, k_1), each server $b \in \{0, 1\}$ evaluates its DPF key k_b on each element of the set $S = \{\sigma_1, \sigma_2, \dots\}$ to produce a vector

$$\bar{v}_b = \big(\mathsf{Eval}(k_b, \sigma_1), \dots, \mathsf{Eval}(k_b, \sigma_{|S|})\big) \quad \in \mathbb{F}^{|S|}.$$

Say that $\bar{v} = \bar{v}_0 + \bar{v}_1 \in \mathbb{F}^{|S|}$ is "valid" if it zero everywhere with a one at a single index (and is "invalid" otherwise). The servers then run a "sketching" protocol to check that $\bar{v}$ is valid.

The protocol should be:

- **Complete.** If $\bar{v}_0 + \bar{v}_1$ is valid, the servers always accept.
- **Sound.** If $\bar{v}_0 + \bar{v}_1$ is invalid, the servers reject almost always.
- **Zero knowledge.** A single malicious server "learns nothing" by running the protocol, apart from the fact that $\bar{v}_0 + \bar{v}_1$ is valid. In particular, the malicious server does not learn the location or value of the non-zero element. We can use a simulation-based definition to formalize this security property.

Existing sketching techniques suffer from two shortcomings:

- *No protection against malicious servers.* Existing sketching protocols for checking that the secret-shared vector $\bar{v}$ has weight one either do not protect client privacy against malicious behavior by the servers [11]. (Techniques that do protect against malicious servers, either have client-to-server communication that grows linearly in the length of the vector being checked, as in Prio [18], or require extra rounds of interaction between the servers and client [7, 28], or require extra non-colluding servers [1, 19].)
- *Weak protection against malicious clients.* A more fundamental—and more subtle—problem in our setting is that these sketching methods do not necessarily prevent a malicious client from influencing the output more than it should, as prior work observes [11].

  As an extreme example, say that the servers' set S consists of a single string σ that is unknown to the clients. An honest client will submit a pair of DPF keys that expand to shares of a vector that contains a one at a *single* coordinate. In contrast, a malicious client can submit a pair of DPF keys that expand to shares of a vector that is one at *every* coordinate. Even if the servers check that their keys expand to shares of a vector of weight one in the singleton set S, the servers will not detect this attack.

  In this way, the malicious client can have more influence on the output than honest clients do.

4.2 New tool: Malicious-secure sketching

Our first contribution of this section is to give a new lightweight protocol that allows the servers to check that they are holding additive shares $\bar{v}_0$ and $\bar{v}_1$ of a vector $\bar{v} \in \{0, 1\}^m \subseteq \mathbb{F}^m$ of weight one (i.e., that has a single non-zero entry), where $\mathbb{F}$ is a prime field. Unlike prior approaches, we protect against malicious misbehavior by either of the two servers, without needing extra interaction with the client nor needing extra servers.

Our idea is to modify a sketching protocol of Boyle et al. [11] (with security against semi-honest servers) to protect it against malicious behavior on the part of the servers. To do so, we have the client encode its vector $\bar{v}$ using a redundant, "authenticated" randomized encoding, inspired by techniques from the

literature on malicious-secure multiparty computation [20, 21]. We construct the encoding in such a way that if either server tampers with the client's vector, the honest servers will reject the client's vector with overwhelming probability. Simultaneously protecting against both malicious clients and a malicious server while minimizing the extra overhead is a delicate balancing act, we discuss below.

Encoding. In our scheme, we have the client choose a random value $\kappa \xleftarrow{R} \mathbb{F}$ and then encode its vector $\bar{v} \in \mathbb{F}^m$ as the pair: $(\bar{v}, \kappa\bar{v}) \in \mathbb{F}^m \times \mathbb{F}^m$. In words: the encoding consists of (a) the vector $\bar{v}$ and (b) the vector $\bar{v}$ scaled by a random value $\kappa \in \mathbb{F}$. The client sends an additive share of this pair $(\bar{v}, \kappa\bar{v})$ to each of the two servers. Since $\bar{v}$ has weight one, both $\bar{v}$ and $\kappa\bar{v}$ are non-zero only at the same single coordinate. The client can then represent each share of this tuple using a single DPF instance with a longer payload.

The client also provides the servers with some correlated randomness, as we discuss below, which the servers use to run a two-party secure computation.

Sketching. The servers receive from the client additive shares of a tuple $(\bar{v}, \bar{v}^*)$. If the client is honest then $\bar{v}^* = \kappa\bar{v}$.

As in the protocol from [11], the servers then jointly sample a uniform random vector $\bar{r} = (r_1, \ldots, r_m) \in \mathbb{F}^m$ and compute $\bar{r}^* = (r_1^2, \ldots, r_m^2) \in \mathbb{F}^m$. (The servers could generate the random vector $\bar{r}$ using a pseudorandom generator, such as AES in counter mode, seeded with a shared secret. Or, for information-theoretic security when $|\mathbb{F}|$ is large, the servers could take $\bar{r} = (r, r^2, r^3, \ldots, r^m)$.)

Now, the servers compute the inner product of these sketch vectors with both the client's data vector $\bar{v}$ *and* their shares of the encoded vector $\bar{v}^*$. That is, for $b \in \{0, 1\}$, server b computes:

$$z_b \leftarrow \langle \bar{r}, \bar{v}_b \rangle \in \mathbb{F}; \quad z_b^* \leftarrow \langle \bar{r}^*, \bar{v}_b \rangle \in \mathbb{F}; \quad z_b^{**} \leftarrow \langle \bar{r}, \bar{v}_b^* \rangle \in \mathbb{F}.$$

Decision. Finally, the servers use a constant-size secure computation to check that the original sketch would have accepted. Letting $z \leftarrow z_0 + z_1$, $z^* \leftarrow z_0^* + z_1^*$, and $z^{**} \leftarrow z_0^{**} + z_1^{**}$, the servers use secure computation to evaluate:

$$(z^2 - z^*) + (\kappa \cdot z - z^{**}) \in \mathbb{F} \tag{1}$$

and check that the output is 0. Note that the first term corresponds to the original sketch verification of [11], and the second term corresponds to checking consistency of the sharing $(\bar{v}, \kappa\bar{v})$.

Intuitively, the second, $\kappa\bar{v}$-computed term will play a protecting role in the servers' verification polynomial: any attempt of a malicious server to launch a conditional failure attack by modifying the sketch z to $(z + \Delta)$ will result in masking the nonzero (possibly sensitive) contribution of the first term by random garbage in the second term, from the corresponding $\kappa\Delta$ term of $\kappa(z + \Delta)$.

We remark that the function (1) on inputs z, z^*, z^{**} as written is not publicly known to the servers, due to the secret client-selected κ term. A natural approach is to provide the servers additionally with secret shares of κ, to be treated as a further input.[1] Instead, we provide a direct approach for the client to enable secure computation of (1) via appropriate correlated randomness.

The idea follows the general approach of Boyle et al. [9], extending Beaver's notion of multiplication triples [5] to more general functions including polynomial evaluation. Here, the client will provide the servers with additive secret shares of random offsets a, b, c, which they will use to publish masked inputs $Z \leftarrow (z + a)$, $Z^* \leftarrow (z^* + b)$, and $Z^{**} \leftarrow (z^{**} + c)$. Then, in addition, the client will provide secret shares of each coefficient in the resulting polynomial that they wish to compute:

$$\begin{aligned}&[(Z - a)^2 - (Z^* - b)] + [\kappa \cdot (Z - a) - (Z^{**} - c)] \\ &\quad = Z^2 + Z^* - Z^{**} + Z[-2a + \kappa] + [a^2 + b - a\kappa + c].\end{aligned}$$

That is, the client will give additive secret shares of $A := [-2a + \kappa]$ and $B := [a^2 + b - a\kappa + c]$. To evaluate, the servers each apply the above polynomial on the publicly known values Z, Z^*, Z^{**}, using their share of each coefficient; this results in additive shares of the desired output.

Security. Given an honest client, the client-aided two-party computation protocol provides security against a malicious server, up to additive attacks on the inputs z, z^*, z^{**} and output of the computation. The latter is irrelevant in regard to client privacy (recall we do not address correctness in the face of a malicious server). As mentioned above, any additive attack on the inputs $(z + \Delta)$, $(z^* + \Delta^*)$, $(z^{**} + \Delta^{**})$ will result in either random garbage output (if $\Delta \neq 0$) or server-predictable output (if $\Delta = 0$).

At the same time, the protocol preserves security against a malicious client. A malicious client has the ability to send invalid values for $\bar{v}, \bar{v}^*$ (supposedly $\kappa\bar{v}$), A, B. However, incorporating these malicious values into the expression evaluated by the servers still results in an analogous polynomial in the servers' secret values $r_1, \ldots, r_m$ as in [11], and application of Schwartz-Zippel similarly implies that any invalid choice of $\bar{v}$ will result in nonzero output evaluation with probability $1 - 2/|\mathbb{F}|$.

Complexity. Altogether, the client must provide: DPF shares of $(\bar{v}, \kappa\bar{v})$, and additive shares of $a, b, c, A, B \in \mathbb{F}$. Since the desired values of a, b, c are independent random field elements, these shares can be compressed (also across levels of the tree) using PRG seeds, which amortizes away their required communication. This results in extra (amortized) $3 \log |\mathbb{F}|$ bits sent to each server, coming from the increased DPF key size (extra $\mathbb{F}$ element for κ-multiplied payload) plus shares of 2 field elements (A, B).

For the sketch verification, the servers must exchange masked input shares of z, z^*, z^{**} in the first round, and then shares of the computed output in a second round. This corresponds to $4 \log |\mathbb{F}|$ bits of communication of each server to the other, split across two rounds.

We provide a more complete treatment of the sketching procedure in the full version of this work [8].

4.3 New tool: Extractable DPFs

As discussed in Section 4.1, there is a second shortcoming to using sketching-based techniques to protect against malicious clients in our setting. The problem is that if the servers only

[1] This approach indeed will work, though requires care to address the servers' ability to provide additive offsets to κ. Our implementation uses a protocol based on this approach, which is slightly less efficient than the one presented here.

sketch the client-provided DPF keys on the strings in the subset S, a cheating client can potentially gain undue influence by having its DPF keys evaluate to 1 on many different strings in $\{0,1\}^n$. The client will evade detection as long as the client's keys evaluate to 1 on only a single point in the subset S.

We address this second problem by giving a refined analysis of our DPF construction, which is based on the state-of-the-art DPF construction of [11]. In that construction, each DPF key has a "public part"—which is identical for both keys—and a "private part"—which differs between the two DPF keys. In the full version of this paper [8], we prove that using this DPF construction, when instantiated in the random-oracle model, and with a large output space, it is computationally infeasible for a client to find malformed DPF keys that (a) have the same public part and (b) represent the sharing of a vector that is 1 at more than one position known to the client. Moreover, it is possible to efficiently extract the position of 1 from the oracle queries made by a malicious client. We term this strengthened type of DPF an "extractable DPF."

This gives the servers a way to check for client misbehavior: the servers can just check that their DPF keys have identical public parts and then conclude that the keys must represent shares of a vector that contains a "1" at a single relevant index, at most.

The technical idea. Working in the random-oracle model [6], where the underlying PRG is a truly random function, we show that any cheating strategy by a client in Gen is restricted in the following sense. Let k_0, k_1 denote the private parts of DPF keys and pp the public part. With high probability, a malicious client that generates DPF keys $(k_0^*, k_1^*, \mathsf{pp}^*)$, and is limited in the number of calls it makes to the random oracle, can find at most one string x such that $\mathsf{Eval}(k_0^*, \mathsf{pp}^*, x) + \mathsf{Eval}(k_1^*, \mathsf{pp}^*, x) = 1$. In contrast, the client can easily generate keys and multiple strings x such that $\mathsf{Eval}(k_0^*, \mathsf{pp}^*, x) + \mathsf{Eval}(k_1^*, \mathsf{pp}^*, x) = 0$, as in a valid key, or $\mathsf{Eval}(k_0^*, \mathsf{pp}^*, x) + \mathsf{Eval}(k_1^*, \mathsf{pp}^*, x)$ is a random value in the (large) output space. However, finding two pairs of keys whose outputs evaluate to "1" in two different known locations is infeasible. Intuitively, the structure of the keys enables the client to fully control a non-zero value at only one location x.

When used in combination with the sketching approach of Section 4.2, this fact essentially implies a complete defense against malicious clients. Indeed, uniqueness of the "1" location means that only this specific vote can be counted, since other nonzero locations will either be caught by the sketching or will not be part of S and therefore not influence the output.

Overall, combining the malicious-secure sketching technique of Section 4.2 with extractable DPFs gives a protocol for private subset histograms that defends privacy against a malicious server and correctness against a malicious client. We note that a similar combination can be useful for other applications of DPF in which the DPF is only evaluated on a strict subset of the input domain. Such applications include private information retrieval by keywords, private distributed storage, and more [11].

The following definition formalizes this notion of extractable DPF in the random-oracle model. Since we envision other applications, we consider here a general (Abelian) output group $\mathbb{G}$, rather than a finite field $\mathbb{F}$. Syntactically, an *extractable DPF* scheme is a DPF scheme $(\mathsf{Gen}, \mathsf{Eval})$ with the modification that the Gen algorithm has an additional output pp (public parameters) that the Eval algorithm takes as an additional input. Our analysis assumes that the input length n, group $\mathbb{G}$, and target nonzero payload β^* ($\beta^* = 1$ by default) are chosen independently of the random oracle.

Definition 1 (Extractable DPF, Simplified)**.** *We say that a DPF scheme in the random-oracle model is* extractable *if there is an efficient extractor E, such that every efficient adversary A wins the following game with negligible probability in the security parameter λ, taken over the choice of a random oracle G and the secret random coins of A.*

- $(1^n, \mathbb{G}, \beta^*) \leftarrow A(1^\lambda)$*, where $\mathbb{G}$ is an Abelian group of size $|\mathbb{G}| \geq 2^\lambda$ and β^* is a nonzero group element.*
- $(k_0^*, k_1^*, \mathsf{pp}^*, x^*) \leftarrow A^G(1^\lambda, 1^n, \mathbb{G}, \beta^*)$*, where $x^* \in \{0,1\}^n$, and G is a random oracle. We assume that pp^* includes the public values $(1^\lambda, 1^n, \mathbb{G})$.*
- $x \leftarrow E(k_0^*, k_1^*, \mathsf{pp}^*, \beta^*, T)$*, where $x \in \{0,1\}^n$ and $T = \{q_1, \ldots, q_t\}$ is the transcript of A's t oracle queries.*

We say that A wins the game if $x^* \neq x$ and $\mathsf{Eval}^G(k_0^*, \mathsf{pp}^*, x^*) + \mathsf{Eval}^G(k_1^*, \mathsf{pp}^*, x^*) = \beta^*$.

Note that in the above definition, the goal of the extractor E is to find the only input x known to A on which the output is β^*. If A could find two or more such inputs, it could win the game with high probability by picking x^* at random from this list. In the full version of this work [8], we define a more general notion of extractability, which applies to incremental DPFs (Section 6) and prove the following claim.

Lemma 4.1 (Informal)**.** *The public-parameter variant of the DPF from [11] is an extractable DPF with winning probability bounded by $\epsilon_A = \left(4t^2 + 2nt + 1\right)/2^\lambda$, where n and t are the length of x^* output by A and number of oracle calls made by A, respectively, and λ is the security parameter. The same holds for the Incremental DPF we construct in Section 6.*

5 Private heavy hitters

We now turn to the problem of collecting *t-heavy hitters* in a privacy-preserving way (Task II of Section 2.1). As before, there are C clients and each client i holds a string $\alpha_i \in \{0,1\}^n$. Now, for a parameter $t \in \mathbb{N}$, the servers want to learn every string that appears in the list $(\alpha_1, \ldots, \alpha_C)$ at least t times.

We first show in Section 5.1, following prior work [3,16,17,48], that the servers can efficiently find all t-heavy hitters by making what we call "prefix-count queries" to the list of client strings $(\alpha_1, \ldots, \alpha_C)$. Next, in Section 5.2, we show how each client i can give the servers a secret-shared encoding of its string α_i that enables the servers to very efficiently make prefix-count queries to the list of client strings $(\alpha_1, \ldots, \alpha_C)$.

The resulting protocol is lightweight: the client sends roughly n PRG keys to each server. When configured to search for t-heavy hitters for $t = \tau C$, the protocol requires server-to-server communication $O(\lambda n C/\tau)$ and server-to-server computation dominated

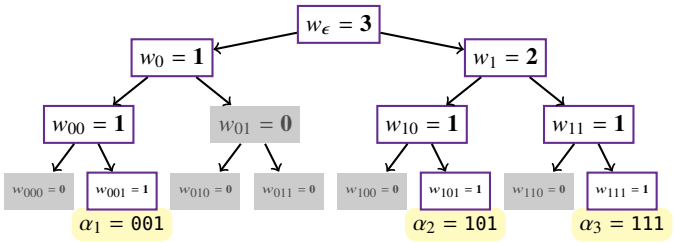

Figure 2: An example prefix tree on strings $(\alpha_1, \alpha_2, \alpha_3)$ of length $n = 3$. The weight w_p of a prefix $p \in \{0,1\}^*$ is the number of strings in the tree that have p as a prefix.

by $O(nC/\tau)$ PRG operations. The protocol requires $O(n)$ rounds of communication.

5.1 Heavy hitters via prefix-count queries

As a first step to understand our approach, imagine that, for any string $p \in \{0,1\}^*$, the servers can make queries of the form:

> How many of the clients' input strings $\alpha_1, \ldots, \alpha_C \in \{0,1\}^n$ start with the prefix p?

We call these "prefix-count queries." For example, suppose there are three clients with strings $(\alpha_1, \alpha_2, \alpha_3) = (\texttt{001}, \texttt{101}, \texttt{111})$. The answer to the query "$p = \epsilon$" (where ϵ is the empty string) would be "3," the answer to the query "$p = \texttt{1}$" would be "2," the answer to the query "$p = \texttt{10}$" would be "1," the answer to the query "$p = \texttt{101}$" would be "1," and the answer to the query "$p = \texttt{01}$" would be "0."

We first show that if the servers can get the answers to such queries, then they can use a simple algorithm to efficiently enumerate all t-heavy hitters among the list of all clients' input strings. This is a classic observation from the literature on streaming algorithms for heavy hitters [16,17], which also appears in recent work on heavy hitters in the local model of differential privacy [3] and in the context of federated learning [48].

This algorithm corresponds to a breadth-first-search of the prefix tree corresponding to the set of strings (Figure 2), in which the search algorithm prunes nodes of weight less than t. To give some intuition for how the algorithm works: let us say that a prefix string $p \in \{0,1\}^*$ is a "heavy" if at least t strings in $\alpha_1, \ldots, \alpha_C \in \{0,1\}^n$ start with p. Then we have the following observations:

- The empty string ϵ is always heavy.
- If a string p is not heavy, then $p\|\texttt{0}$ and $p\|\texttt{1}$ are not heavy.
- If a string p is heavy and p is n characters long (i.e., $|p| = n$), then p is a t-heavy hitter.

These three observations immediately give rise to Algorithm 3. For each prefix length $\ell \in \{0, \ldots, n\}$, we construct the set H_ℓ of heavy strings of length ℓ. The set H_0 consists of the empty string ϵ, since ϵ is always heavy (assuming, without loss of generality that $t \leq C$). We construct the set H_ℓ by appending $\texttt{0}$ and $\texttt{1}$ to each element of $H_{\ell-1}$ and checking whether the resulting string is heavy. And finally, H_n consists of all of the t-heavy hitters.

Efficiency. The clients have C strings total. Then, for for any string length $\ell \in \{0, \ldots, n\}$, there are at most C/t heavy strings of length ℓ. At each level ℓ, the algorithm of Algorithm 3 makes at most one oracle query for each heavy string. The algorithm thus makes at most $n \cdot C/t$ prefix-count-oracle queries total. If we are looking strings that more than a constant fraction of all clients hold (e.g., $t = 0.001C$), then the number of queries will be independent of the number of clients.

Algorithm 3: t-heavy hitters from prefix-count queries. The algorithm is parameterized by a string length $n \in \{0,1\}$ and a threshold $t \in \mathbb{N}$.

Input: The algorithm has no explicit input, but it has access to a "prefix-count" oracle $\mathcal{O}_{\alpha_1,\ldots,\alpha_C}$. For any string $p \in \{0,1\}^*$, the oracle $\mathcal{O}_{\alpha_1,\ldots,\alpha_C}(p)$ returns the number of strings in $(\alpha_1, \ldots, \alpha_C)$ that begin with prefix p.

Output: The set of all t-heavy hitters in $(\alpha_1, \ldots, \alpha_C)$.

Algorithm.

- Let $H_0 \leftarrow \{\epsilon\}$ (a set containing the empty string).
- Let $w_\epsilon \leftarrow C$.
- For each prefix length $\ell = 1, \ldots, n$:
  - Let $H_\ell \leftarrow \emptyset$.
  - For each prefix $p \in H_{\ell-1}$:
    - $w_{p\|\texttt{0}} \leftarrow \mathcal{O}_{\alpha_1,\ldots,\alpha_C}(p\|\texttt{0})$, and
    - $w_{p\|\texttt{1}} \leftarrow w_p - w_{p\|\texttt{0}} \in \mathbb{Z}$.

    Then:
    - If $w_{p\|\texttt{0}} \geq t$, add $p\|\texttt{0}$ to H_ℓ.
    - If $w_{p\|\texttt{1}} \geq t$, add $p\|\texttt{1}$ to H_ℓ.
- Return H_n.

Security and leakage. While searching for the heavy hitters, the servers will learn more information than just the t-heavy hitters themselves. In particular, the servers will learn:

(a) the set of all heavy strings and
(b) for every heavy string p, the number of strings in $(\alpha_1, \ldots, \alpha_n)$ that begin with p.

As we discuss in Section 7, it is possible to apply ideas from differential privacy to limit the damage that either type of the leakage can cause.

5.2 Implementing private prefix-count queries via incremental DPFs

Given the techniques of Section 5.1, we now just need to explain how the servers can compute the answers to prefix-count queries over the set of client-held strings without learning anything else about the clients' input strings.

We do this using *incremental distributed point functions*, a new cryptographic primitives that builds on standard distributed point functions (DPFs, introduced in Section 3). Using standard DPFs for our application would also work, but would be more expensive, both asymptotically and concretely. If each client holds an n-bit string, with plain DPFs, the client computation and communication costs would grow as n^2. With incremental

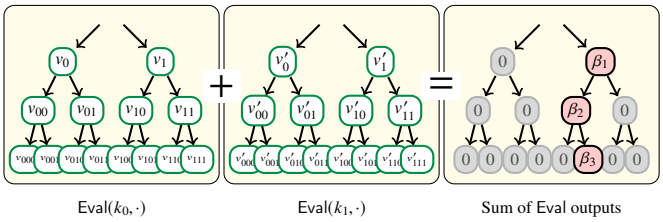

Figure 4: Incremental DPFs give concise secret-sharing of the values on the nodes of a tree, such that the tree contains a single non-zero path. In this example, the depth $n = 3$, the special point $\alpha = \texttt{101}$, the values on the path are $\beta_1 \in \mathbb{G}_1, \beta_2 \in \mathbb{G}_2, \beta_3 \in \mathbb{G}_3$ for some finite groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_3$, and the keys are generated as $\mathsf{Gen}(\alpha, \beta_1, \beta_2, \beta_3) \to (k_0, k_1)$.

DPFs, this cost falls to linear in n. For our applications, $n \approx 256$, so this factor-of-n performance improvement is substantial. We get similar performance improvements on the server side.

We first define incremental DPFs, then use them to solve the heavy-hitters problem, and finally explain how to construct them.

New tool: Incremental DPF. A standard distributed point function gives a way to succinctly secret share a *vector* of dimension 2^n that is non-zero at a single point. By analogy, we can think of an incremental DPF as a secret-shared representation of the values on the nodes of a binary *tree* with 2^n leaves, where there is a single non-zero path in the tree whose nodes have non-zero values (Figure 4).

More precisely, an incremental DPF scheme, parameterized by finite groups $\mathbb{G}_1, \ldots, \mathbb{G}_n$, consists of two routines:

- $\mathsf{Gen}(\alpha, \beta_1, \ldots, \beta_n) \to (k_0, k_1)$. Given a string $\alpha \in \{0,1\}^n$ and values $\beta_1 \in \mathbb{G}_1, \ldots, \beta_n \in \mathbb{G}_n$, output two keys.

  We can think of the incremental DPF keys as representing secret shares of the values on the nodes of a tree with 2^n leaves and a single non-zero path. Using this view, $\alpha \in \{0,1\}^n$ is the index of the leaf at the end of the non-zero path. The values $\beta_1, \ldots, \beta_n$ specify the values that the nodes along the non-zero path take. (For simplicity, we do not assign a value to the root node of the tree. This is without loss of generality.)
- $\mathsf{Eval}(k, x) \to \cup_{\ell=1}^{n} \mathbb{G}_\ell$. Given an incremental DPF key k and string $x \in \bigcup_{\ell=1}^{n} \{0,1\}^\ell$, output a secret-shared value.

  If we take the view of incremental DPF keys as shares of the values of the nodes on a binary tree, $\mathsf{Eval}(k, x)$ outputs a secret sharing of the value on the xth node of the tree, where we associate each node in the tree with a bitstring in $\bigcup_{\ell=1}^{n} \{0,1\}^\ell$ in the natural way.

The incremental DPF correctness property states that, for all strings $\alpha \in \{0,1\}^n$, output values $\beta_1 \in \mathbb{G}_1, \ldots, \beta_n \in \mathbb{G}_n$, keys $(k_0, k_1) \leftarrow \mathsf{Gen}(\alpha, \beta)$, and values $x \in \bigcup_{\ell=1}^{n} \{0,1\}^\ell$, it holds that

$$\mathsf{Eval}(k_0, x) + \mathsf{Eval}(k_1, x) = \begin{cases} \beta_\ell & \text{if } |x| = \ell \text{ and} \\ & x \text{ is a prefix of } \alpha \\ 0 & \text{otherwise} \end{cases},$$

where $|x| = \ell$ and the addition is computed in the finite group $\mathbb{G}_\ell$. Informally, the DPF security property states that an adversary that learns either k_0 or k_1 (but not both) learns no information about the special point α or the values $\beta_1, \ldots, \beta_n$.

We can use standard DPFs in a black-box way to build incremental DPFs: we secret share the values at each of the n levels of the tree using a single pair of DPF keys. With state-of-the-art DPFs, the resulting construction has key size and evaluation time proportional to n^2, if $\alpha \in \{0,1\}^n$.

In contrast, we give a direct construction of incremental DPFs from pseudorandom generators (PRGs) that has essentially optimal key size and evaluation time. More specifically, each incremental DPF key has bitlength $O(\lambda \cdot n) + \sum_{\ell=1}^{n} \log_2 |\mathbb{G}_\ell|$, when instantiated with a length-doubling PRG that uses λ-bit keys and achieves $\Omega(\lambda)$-bit security. We describe our construction in Section 6.

Using incremental DPFs to implement heavy hitters. We now explain how to build a system for computing t-private heavy hitters using incremental DPFs (Protocol 5).

At a high level, each client i produces a pair of incremental DPF keys that represent the secret sharing of a prefix tree that is zero everywhere, but whose nodes have value 1 on the path down to client i's input string α_i.

Given incremental DPF keys from all C clients, the two servers can compute the answers to prefix-count queries by publishing a single message each. To compute the number of client strings that start with a prefix $p \in \{0,1\}^*$, each server evaluates all of the clients' incremental DPF keys on the prefix p and outputs the sum of these evaluations.

Using this technique, the servers can run the protocol of Algorithm 3 to find all of the t-heavy hitters.

Efficiency. The client-to-server communication consists of a single incremental DPF key. The server-to-server communication requires a number of field elements proportional to the number of prefix-count oracle queries that the servers make. As we argued in Section 5.1, this is at most $n \cdot C/t$.

Semi-honest security. If all parties (clients and servers) follow the protocol, then a semi-honest adversary controlling one of the two servers learns no more about the client strings $(\alpha_1, \ldots, \alpha_C)$ that what the servers learn from the heavy-hitters algorithm of Algorithm 3. Section 7 discusses how to use ideas from differential privacy to ameliorate the effects of this leakage. In principle, it also would be possible for the servers to use a constant-sized secure two-party computation [47] to reduce the leakage to a single bit per prefix-count oracle query. Since this approach is substantially more complicated to implement, and since our protocol's leakage is already quite modest, we do not discuss this direction further.

In practice, clients and servers may deviate from the protocol, which we discuss here:

Protection against malicious clients. As in Section 4, malicious clients can submit malformed incremental DPF keys with the goal of corrupting or over-influencing the output of the protocol. We can protect against malicious clients here using our tools from Section 4.

In particular, the servers will run the protocol of Protocol 5, instantiated with the t-heavy-hitters algorithm of Algorithm 3. In this protocol, for each prefix length $\ell = 1, \ldots, n$, the servers assemble a set—call it S_ℓ—of candidate heavy prefixes of length

Protocol 5: Private t-heavy hitters (semi-honest secure version). Our full protocol uses sketching to achieve security against malicious clients (Section 4). We elide the sketching step here for clarity. There are two servers and C clients. Each client i, for $i \in [C]$, holds a string $\alpha_i \in \{0,1\}^n$. The servers want to learn the set of all t-heavy hitters in $(\alpha_1, \ldots, \alpha_C)$. The incremental DPF works over the additive group of a finite field $\mathbb{F}$ where $|\mathbb{F}| > C$. The protocol is as follows:

1. Each client $i \in \{1, \ldots, C\}$, on input string $\alpha_i \in \{0,1\}^n$, sets $\beta_1 = \cdots = \beta_n = 1 \in \mathbb{F}$ and prepares a pair of incremental DPF keys as
$$(k_0^{(i)}, k_1^{(i)}) \leftarrow \mathsf{Gen}(\alpha_i, \beta_1, \ldots, \beta_n).$$
The client sends key $k_0^{(i)}$ to Server 0 and key $k_1^{(i)}$ to Server 1. After sending this single message to the servers, Client i can go offline.
2. The servers jointly run Algorithm 3. Whenever that algorithm makes a prefix-count oracle query on a prefix string $p \in \{0,1\}^*$, each server $b \in \{0,1\}$ computes and publishes the value
$$\mathsf{val}_{p,b} \leftarrow \sum_{i=1}^{C} \mathsf{Eval}(k_b^{(i)}, p) \quad \in \mathbb{F}.$$
Both servers recover the answer to the prefix-count oracle query as
$$\mathsf{val}_p \leftarrow \mathsf{val}_{p,0} + \mathsf{val}_{p,1} \quad \in \mathbb{F}.$$
3. The servers output whatever the algorithm of Algorithm 3 outputs.

ℓ. The servers will then evaluate all of the clients' incremental DPF keys at these points.

If the client is honest, the incremental DPF keys evaluated at the points in S_ℓ will be shares of a vector that is zero everywhere with a one at at most a single position. Specifically, for prefix length ℓ, client i's incremental DPF keys should evaluate to shares of the value "1" on the ℓ-bit prefix of client i's string α_i. The keys should evaluate to zero everywhere else.

So now the servers have the same task as in Section 4: the servers hold secret shares of a client-provided vector and the servers want to check that this vector is zero everywhere except that it is "1" at at most a single coordinate. Thus, to prevent misbehavior my malicious clients, at each level $\ell \in [n]$ of the tree, the servers can use our malicious-secure sketching schemes from Section 4 to check that this property holds. At each level of the tree, for each client, the servers perform one round of malicious-secure sketching.

We use the malicious-secure sketching approach of Section 4.2, which requires the client to encode its data using a redundant randomized encoding.

Full security: Protection against malicious servers. Our final task is to analyze the security of the protocol of Protocol 5 against actively malicious behavior by one the two participating servers.

A malicious server's only strategy to learn extra information in Protocol 5 is to manipulate answers to the prefix-count oracle queries using an "additive attack." For example, in Step 2 of the protocol, in processing the answer to a prefix-oracle query p, Server 0 is supposed to publish $\mathsf{val}_{p,0} = \sum_{i=1}^{C} \mathsf{Eval}(k_0^{(i)}, p)$. If the server is malicious, it could instead publish the value $\mathsf{val}_{p,0} + \Delta$, for some non-zero shift $\Delta \in \mathbb{F}$.

We capture the power of this attack in our formal security definitions (the full version of this work [8]), which quantify the information that the adversary can learn from such additive attacks. Intuitively: the adversary can essentially control which strings are heavy hitters (and can thus learn how many honest clients hold strings in a small set), but the adversary can do not much worse than this. As we discuss in Section 7, it is possible to further limit the power of this leakage using differential privacy.

Extension: Longer strings. The techniques outlined so far allow for the private computation of t-heavy hitters over n-bit strings in which each client sends each server an all-prefix DPF key with domain size n. Each key is roughly $\lambda n \log_2 C$ bits in length, where C is the number of participating clients and $\lambda \approx 128$ is the size of a PRG seed.

In some applications, the servers might want to compute the most popular values over relatively long strings. For example, an operating-system vendor might want to learn the set of popular software binaries running on clients' machines that touch certain sensitive system files. In this application, client i's string $\alpha_i \in \{0,1\}^n$ is an x86 program, which could be megabytes long. So for this application, $n \approx 2^{20}$.

When n is much bigger than λ, we can use hashing to reduce the client-to-server communication from $\approx \lambda n \log_2 C$ bits down to $\approx \lambda^2 \log_2 C + n$ bits and the round complexity from $\approx n$ to $\approx \lambda$. We describe this extension in the full version of this work [8].

6 Constructing Incremental DPFs

A straightforward way to construct an incremental DPF would be to generate n independent distributed point function (DPF) keys, one for each prefix length, and to evaluate $x \in \{0,1\}^\ell$ using the ℓ-th key. Given the most efficient DPF solution [11], this would yield overall key size and computation for all-prefix evaluation (in units of PRG invocations) both *quadratic* in the input bit length n. In contrast, our goal is to construct a more efficient scheme for all-prefix DPF in which all these measures are linear in n. We achieve precisely this goal, leveraging the specific structure of the DPF construction of [11].

We give the formal syntax and definitions in the full version of this work [8]. (See Section 5.2 for informal definitions.) In the remainder of this section, we sketch our construction of incremental DPF.

Construction. We construct an efficient incremental DPF scheme, whose key size and generation/evaluation computation costs in particular grow *linearly* with the input bit length n.

In the (standard) DPF construction of [11], the evaluation of a shared point function $f_{\alpha,\beta}(x) : \{0,1\}^n \to \mathbb{G}_n$ traverses a path defined by the binary representation of x. The procedure generates a pseudo-random value for each node of the path and an element of the output group $\mathbb{G}_n$ at the termination of the path. The two matching DPF keys are set up so that the pseudo-random

	Key size		AES operations	
	Any n	$n = 256$	Any n	$n = 256$
DPF [11]	$\approx \frac{n^2\lambda}{2}$	543 KB	$\approx \frac{n^2}{2}$	32,640 ops.
This work	$\approx n(\lambda + m)$	6.2 KB	$\approx 2n$	513 ops.

Table 6: A comparison of the key size and evaluation time of two alternatives for constructing incremental DPF: using state-of-the-art DPF as a black box and the incremental DPF construction in this paper. In all entries of the table the input length is n, the PRG seed length is $\lambda = 127$, the group size in intermediate levels of the tree is 2^m, $m = 62$, and the group size in the leaves is $2^{2\lambda}$, which suffices for the extractable DPF feature. For asymptotic expressions we assume $m \leq \lambda$. The exact key size in the DPF-based construction is $\frac{n(n+1)(\lambda+2)}{2} + n(\lambda + m) + 2\lambda$ and in the direct incremental DPF construction the key size is $n(\lambda + m + 2) + 4\lambda - m$.

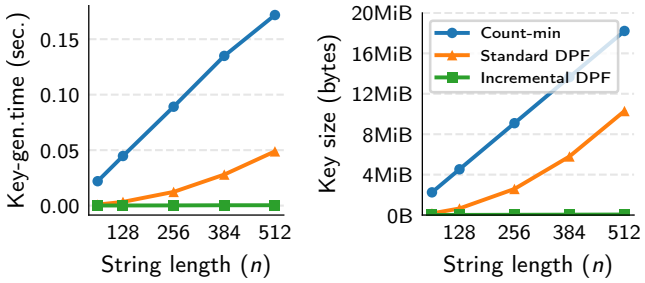

Figure 7: Client-side costs in time (left) and communication (right) of our private heavy-hitters scheme based on our new incremental DPFs, on standard DPFs, and on linear sketching techniques [40]. Costs for our scheme are relatively small and grow linearly in the length n of each client's string.

value generated by the first key is sampled independently of the value generated by the other key, for every prefix of x which is also a prefix of α. However, when the paths to x and α diverge, the evaluation procedure programs the two pseudo-random values to be *equal*, by using extra information encoded in the keys, which we refer to as Correction Words (CW). The evaluation procedure on two identical pseudo-random values generates identical values along the path to x, and the same group value for the output, ensuring that the output is 0 if $x \neq \alpha$. However, if $x = \alpha$ then the two independent pseudo-random values, which are known at key generation time, can be corrected to share the desired output β.

We extend the DPF construction of Boyle et al. [11] to further support prefix outputs with small overhead. The main observation is that the intermediate pseudo-random values generated at each level of DPF evaluation satisfy the same above-described property necessary for the final output level: namely, also for a *prefix* $(x_1, \ldots, x_\ell) \neq (\alpha_1, \ldots, \alpha_\ell)$ the intermediate evaluation generates identical pseudo-random values and for $(x_1, \ldots, x_\ell) = (\alpha_1, \ldots, \alpha_\ell)$ it generates independent pseudo-random values. These pseudo-random values cannot be used directly to share desired intermediate outputs, as this would compromise their pseudo-randomness required for security of the remaining DPF scheme (roughly, using them twice as a one-time pad). Instead, we introduce an extra intermediate step at each level ℓ, which first expands the intermediate pseudo-random value $\tilde{s}^{(\ell)}$ to two pseudo-random values: a new seed $s^{(\ell)}$ which will take the place of $\tilde{s}^{(\ell)}$ in the next steps of the DPF construction, and an element of the ℓth level output group $\mathbb{G}_\ell$ which will be used to generate shares of the desired ℓth output $\beta_\ell \in \mathbb{G}_\ell$.

Ultimately, the new procedure introduces an extra PRG evaluation and group operation per level ℓ, as well as an additional element $W_{CW}^{(\ell)}$ of the ℓth level group $\mathbb{G}_i$ within the key, to provide the desired pseudo-random to target output correction.

The full pseudocode of our incremental DPF construction appears in the full version of this work [8]. It yields the following result:

Proposition 1 (Incremental DPF)**.** *The incremental DPF scheme described in the full version of this work [8] is a secure Incremental DPF with the following complexities for* $(\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n))$*:*

- *Key size:* $\lambda + (\lambda + 2)n + \sum_{j \in [n]} \lceil \log |\mathbb{G}_j| \rceil$ *bits.*
- *Computation: Let* $\mathsf{cost}(\ell) := 1 + \lceil \log(|\mathbb{G}_\ell|)/\lambda \rceil$*. Units given in evaluation of a PRG* $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$*:*
  - Gen*:* $2\sum_{\ell \in [n]} \mathsf{cost}(\ell)$
  - Eval(x)*:* $\sum_{\ell \in [|x|]} \mathsf{cost}(\ell)$

We prove Proposition 1 in the full version of this work [8].

In the full version of this work [8], we describe a number of low-level optimizations that we have implemented to make our incremental DPF construction more efficient, especially when using AES hardware instructions to implement the PRG.

7 Providing differential privacy

To bound the amount of information that an adversary can infer from the system's output, we can ensure that the system's output satisfies ϵ*-differential privacy* [24, 25]. This is possible with a simple tweak to our heavy-hitters protocol (Protocol 5), which we describe in the full version of this work [8].

8 Implementation and evaluation

We implemented our complete private heavy-hitters scheme in Rust (1.46.0-nightly). Our implementation is roughly 3,500 lines of code, including tests. The code is online at https://github.com/henrycg/heavyhitters.

Our sketching scheme uses a 62-bit finite field in the middle of the "tree" (Figure 4) and a 255-bit field at the leaves. With this configuration, our sketching schemes detect cheating clients, except with probability $\approx 2^{-62}$, over the servers' random choices, independent of how much computation a cheating client does. While we expect this level of security against a cheating client to be sufficient in practice, by running the sketching scheme twice—at most doubling the communication and computation—we can achieve nearly 128-bit security. Using the larger field at the leaves ensures that our DPF construction satisfies the extractability property (Lemma 4.1) against cheating clients that run in time at most $\approx 2^{128}$.

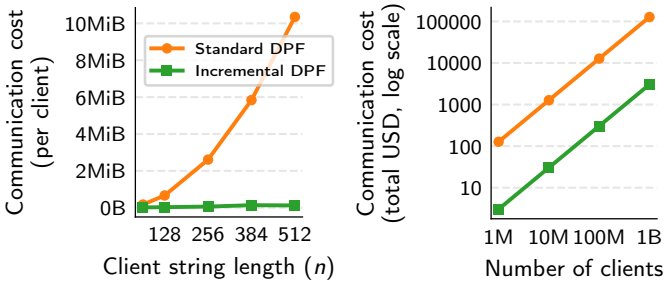

Figure 8: Total server communication cost (send + receive) per client for our private heavy-hitters scheme. At left: we simulate the server-side communication cost our scheme and compare to a scheme based on standard DPFs. At right: we compare the US-dollar cost of the schemes used with 256-bit strings searching for the top-900 heavy hitters, using Amazon EC2's current (Dec. 2020) data-transfer price of USD 0.05/GB.

Client costs. Figure 7 shows the client costs for three different private heavy-hitters schemes. Our client experiments run on an Intel i7-1068NG7 CPU at 2.3 GHz. The first is our tree-based scheme (Section 5), based on our new incremental DPFs (Section 6). The second uses our tree-based scheme, but with standard DPFs [11]. The third uses private aggregation of count-min sketches [17, 40] to compute approximate heavy hitters. For the count-min sketches, we set the approximation error $\epsilon = 1/128$ and failure probability $\delta = 2^{-60}$. (To reduce communication in this third scheme, we use DPFs here as well.)

Our incremental DPF keys have size linear in the length of the clients' strings, with a small constant. In contrast, using standard DPFs requires one linear-sized key for each layer of the prefix tree (Section 5), which yields a quadratic cost. The count-min-sketch based private aggregation scheme also has a linear client-side cost, but the large size of each sketch makes the constant substantially worse.

Server communication. Figure 8 shows the total communication cost per server per client of running our end-to-end heavy hitters protocol. In this experiment, clients sample their strings from a Zipf distribution with parameter 1.03 and support 10,000. This type of "power-law" distribution arises naturally in network settings [38] and we choose the parameter conservatively (i.e., the distribution is closer to uniform than we would expect in nature), which likely gives an underestimate of our system's performance. In this experiment, servers search for strings that more than 0.1% of clients hold. In our schemes, the total communication per client is tens of kilobytes. Figure 8 also estimates the dollar cost of computing private heavy hitters using the baseline scheme (based on standard DPFs) and our scheme, as the number of clients varies. Our scheme is roughly two orders of magnitude less expensive.

End-to-end performance. Finally, we ran an end-to-end performance test of the system over the Internet. We use one c4.8xlarge server (32 virtual cores) in Amazon's us-east-1 region (N. Virginia) and one in the us-west-1 region (N. California). The round-trip latency between the two data centers was 61.8ms. We measure the running time from the moment after the servers collect the last incremental DPF keys from the clients until the servers produce their output. Each client holds a 256-bit string,

	Running time (sec.)			
Clients	DPF	Sketching	Total	Clients/Sec.
100k	107.3	704.5	828.1	120.8
200k	211.0	1,404.1	1,633.5	122.4
400k	433.5	2,771.4	3,226.0	124.0

Table 9: End-to-end cost of our private heavy-hitters system, used to collect $n = 256$-bit strings. Each client's string is sampled from a Zipf distribution with parameter 1.03 and support 10,000. We use one c4.8xlarge server (32 virtual cores) to implement each of the two logical servers. One server is in Amazon's N. California data center and the other is in N. Virginia. The servers set the heavy-hitters threshold to collect all strings that more than 0.1% of clients hold.

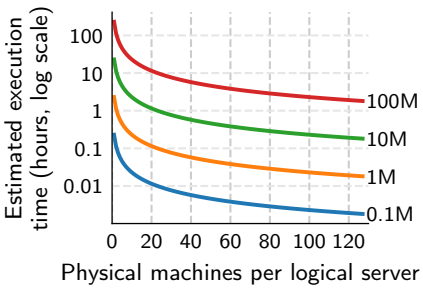

Figure 10: Estimated execution time of our heavy-hitters protocol. Each line represents a number of clients. The workload is fully parallelizable, so we neglect sharding costs. System parameters are as in Table 9.

which is enough to represent a 42-character domain name (uncompressed). Table 9 shows the results of this experiment. For 400,000 clients, the total running time is around 53 minutes.

Our heavy-hitters protocol is almost completely parallelizable. In Figure 10, we give estimates for the protocol-execution time, as a function of the number of clients and the number of physical machines used to implement each of the system's two logical servers. When deployed with 20 machines per logical server, we estimate that the system could process ten million client requests in just over one hour.

9 Conclusions

In this paper, we have described a system that allows two non-colluding servers to compute the most popular strings among a large set of client-held strings in a manner that preserves client privacy. Along the way, we introduce several lightweight cryptographic tools: a protocol for checking that a secret-shared vector is a unit vector, an extractable variant of distributed point functions that defends against badly formed keys, and a generalization of distributed point functions for secret-sharing weights on binary trees.

There are a number of potential extensions to this work. For instance, instead of finding heavy hitters, the servers might like to find heavy *clusters*—strings that are close to many of the client-held strings, under some distance metric. Perhaps each client holds a GPS coordinate pair and the servers want to learn the popular neighborhoods.

Acknowledgments. We thank Eric Rescorla for suggesting this problem, Saba Eskandarian for comments and discussion, and the anonymous reviewers for their feedback and suggestions. Dan Boneh was funded by NSF, DARPA, a grant from ONR, and the Simons Foundation. Elette Boyle was supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC Project HSS (852952). Henry Corrigan-Gibbs was funded in part by a Facebook Research Award. Henry thanks Bryan Ford for generously hosting him at EPFL during the early stages of this project. Niv Gilboa was supported by ISF grant 2951/20, ERC grant 876110, and a grant by the BGU Cyber Center. Yuval Ishai was supported by ERC Project NTSC (742754), ISF grant 2774/20, NSF-BSF grant 2015782, and BSF grant 2018393.

References

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: MPC based scalable and robust anonymous committed broadcast., 2020.

[2] Benny Applebaum, Haakon Ringberg, Michael J Freedman, Matthew Caesar, and Jennifer Rexford. Collaborative, privacy-preserving data aggregation at scale. In *PETS*, pages 56–74. Springer, 2010.

[3] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. In *Neural Information Processing Systems*, pages 2288–2296, 2017.

[4] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 127–135, 2015.

[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432. Springer, 1991.

[6] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.

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

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. arXiv, December 2020.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019*, pages 341–371.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

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

[12] Mark Bun, Jelani Nelson, and Uri Stemmer. Heavy hitters and the structure of local privacy. *ACM Transactions on Algorithms (TALG)*, 15(4):1–40, 2019.

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

[14] T-H Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Transactions on Information and System Security (TISSEC)*, 14(3):1–24, 2011.

[15] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[16] Graham Cormode, Flip Korn, Shanmugavelayutham Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, 2003.

[17] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

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

[19] 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.

[20] Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors. In *EUROCRYPT*, pages 471–488, 2008.

[21] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

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

[23] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535, 2017.

[24] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[25] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

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

[27] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

[28] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. *arXiv preprint arXiv:1911.09215*, 2019.

[29] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a Rappor with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies*, 2016(3):41–61, 2016.

[30] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In *FOCS*, pages 210–229. IEEE, 2015.

[31] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658, 2014.

[32] O Goldreich, S Micali, and A Wigderson. How to play any mental game. In *STOC*, pages 218–229, 1987.

[33] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[34] S Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 513–524, 2012.

[35] Rob Jansen and Aaron Johnson. Safely measuring Tor. In *CCS*, pages 1553–1567, 2016.

[36] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *PETS*, pages 221–238. Springer, 2012.

[37] Marcel Keller and Avishay Yanai. Efficient maliciously secure

multiparty computation for RAM. In *EUROCRYPT*, pages 91–124. Springer, 2018.

[38] Jon Kleinberg and Steve Lawrence. The structure of the web. *Science*, 294(5548):1849–1850, 2001.

[39] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference*, pages 377–396. Springer, 2013.

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

[41] Mozilla. Your browser is hijacked, now what? https://blog.mozilla.org/firefox/your-browser-is-hijacked-now-what/, Accessed 19 August 2020, October 2018.

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

[43] Raluca Ada Popa, Hari Balakrishnan, and Andrew J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security*, pages 335–350, 2009.

[44] 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.

[45] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[46] David Isaac Wolinsky, Henry Corrigan-Gibbs, Aaron Johnson, and Bryan Ford. Dissent in numbers: Making strong anonymity scale. In *10th OSDI*. USENIX, October 2012.

[47] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.

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