PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries with Full Security

Dimitris Mouris[1*], Pratik Sarkar[2*], and Nektarios Georgios Tsoutsos[1]

[1] University of Delaware
{jimouris, tsoutsos}@udel.edu
[2] Boston University
pratik93@bu.edu

Abstract. The private heavy-hitters problem is a data-collection task where many clients possess private bit strings, and data-collection servers aim to identify the most popular strings without learning anything about the clients' inputs. The recent work of Poplar constructed a protocol for private heavy hitters but their solution was susceptible to additive attacks by a malicious server, compromising both the correctness and the security of the protocol.

In this paper, we introduce PLASMA, a private analytics framework that addresses these challenges by using three data-collection servers and a novel primitive, called verifiable incremental distributed point function (VIDPF). PLASMA allows each client to non-interactively send a message to the servers as its input and then go offline. Our new VIDPF primitive employs lightweight techniques based on efficient hashing and allows the servers to non-interactively validate client inputs and preemptively reject malformed ones.

PLASMA drastically reduces the communication overhead incurred by the servers using our novel batched consistency checks. Specifically, our server-to-server communication depends only on the number of malicious clients, as opposed to the total number of clients, yielding a 182× and 235× improvement over Poplar and other state-of-the-art sorting-based protocols respectively. Compared to recent works, PLASMA enables both client input validation and succinct communication, while ensuring full security. At runtime, PLASMA computes the 1000 most popular strings among a set of 1 million client-held 32-bit strings in 67 seconds and 256-bit strings in less than 20 minutes respectively.

Keywords: Function secret sharing, private histograms, private heavy hitters, secure multiparty computation

[*] The first two authors have equal contribution and appear in alphabetical order.
D. Mouris and N.G. Tsoutsos would like to acknowledge the support of the Electrical & Computer Engineering department and the College of Engineering at the University of Delaware for supporting this research.
P. Sarkar is supported by NSF Awards 1931714, 1414119, and the DARPA SIEVE program.

Table of Contents

1 Introduction

In today's technology-driven world, companies are constantly collecting user data to perform data analysis, compute statistics, expose patterns in user behaviors, and apply them to improve their products [33, 41, 19, 46]. Common practices for data analytics resort to histograms, where client data are aggregated together in predefined and non-overlapping buckets. Each bucket may represent a quantitative range (e.g., salary) or a categorical value (e.g., profession). The resulting histogram displays the frequencies of each bucket based on multiple aggregated participant (or client) responses.

Private Histograms. When computing statistics using histograms, it is crucial to maintain client privacy, such as preventing data collection servers from inferring additional information about clients' inputs. Existing solutions for privacy-preserving histograms can solve this problem efficiently, given a relatively small number of buckets [26, 8, 29, 53, 2, 11]. Nevertheless, histograms tend to be resource-intensive on the server side when the goal is to find the most popular entries among the clients' inputs. For instance, assume clients that hold GPS coordinates of their location and servers aiming to discover crowded areas without compromising client privacy. The naive solution of creating a histogram over all possible inputs results in sparsely populated sets over all possible inputs, which wastes server-side computational power due to sparse inputs. Conversely, in an optimal solution the server computation should scale with the most popular inputs (instead of all possible inputs).

Private Heavy-Hitters. This problem is addressed by the concept of "heavy hitters". $\mathcal{T}$-heavy hitters allow computing the $\mathcal{T}$ most popular responses (for a given threshold $\mathcal{T}$) among clients' inputs and have a broad range of applications: from finding popular websites that users visit or malicious URLs that cause browsers to crash [40, 16], to discovering commonly used passwords [55], learning new words typed by users and identifying frequently used emojis [34], to name a few. Private heavy-hitters allow computing these results while also preserving client privacy. Existing protocols (such as [25, 57, 55, 16, 2, 15]) only focus on the "popular" inputs and disregard other inputs that appear less than $\mathcal{T}$ times (i.e., they are pruned by the protocol). This renders private heavy-hitters a suitable candidate for finding the most common client entries, such as computing crowded areas using client-provided GPS coordinates.

Different Approaches. The literature considers the setting where two or more servers collect client inputs and run the private heavy-hitters protocol among these servers. A notable approach is based on differential privacy (DP), and the current state-of-the-art is [2]; we discuss more DP-based solutions in our related works (Section 1.2). While these protocols are computationally fast, an important drawback is that they are limited to DP-based privacy guarantees for the client. Likewise, MPC-based solutions (such as [15]) employ general-purpose secure computation frameworks (e.g., MP-SPDZ [44], SCALE-MAMBA [1], Sharemind [14]), but these methods fall short of practical expectations. Thus, recent works introduced customized MPC-based techniques to solve the private heavy-hitters problem [5, 42]. The underlying protocols perform secure sorting of client inputs under MPC [39, 9, 13, 4] and then aggregate the sorted data. This guarantees that the clients' inputs remain hidden when a majority of the servers are honest. However, all the aforementioned solutions incur server-to-server communications that scale linearly with the total number of clients, which is unfavorable when this number increases.

Distributed point functions (DPF) [37, 17, 18] offer an alternative approach for private histograms. Informally, DPF allows a client to generate and send succinct shares of a point function corresponding to their private inputs to two servers. The servers then use these shares to locally evaluate the point function on every possible input in the entire input space and add the resulting outputs to obtain additive shares of a histogram. However, this approach incurs quadratic (i.e., $\mathcal{O}(n^2)$) client-to-server communication (where n is the number of bits required to represent a client's input) for privately computing heavy hitters.

Poplar. Recent work in Poplar [16] extends the DPF approach by introducing the notion of incremental DPF (IDPF) (more details in Section 2.3). Poplar provides an IDPF-based solution for the private heavy-hitter in the two-server setting, which reduces the quadratic client-server communication of DPF-based solutions to linear. Their server-to-server communication depends only on the input string length in the semi-honest setting. For security against malicious clients, the servers validate every client's input so that malformed inputs are preemptively detected by the servers and discarded from the computation. This is referred to as *client input validation* and it prevents a malicious client from causing an abort in the entire protocol involving other clients. Poplar requires additional checks to perform input validation against malicious clients, which results in the server-to-server communication to scale linearly with the total number of participating clients.

Motivation. Since all aforementioned solutions incur server-to-server communication that scales linearly with the number of clients, they are prohibitive for most real-world applications that require millions of participating clients for data collection. Likewise, neither Poplar nor the DP-based solutions [15, 2] can tolerate additive attacks from a malicious server, which results in incorrect outputs when one of the servers do not follow the protocol steps. More formally, they fail to provide *full security* (i.e., both correctness and privacy) against the collusion of a malicious server and malicious clients. In this regard, we summarize the above solutions in Table 1 and ask the following motivating question:

Can we obtain a protocol for private heavy hitters that guarantees full security with succinct server-to-server communication?

1.1 Our Contributions

We answer the aforementioned research question by presenting PLASMA, a framework for private and lightweight statistics that provides full security (against a malicious server + against malicious clients) while maintaining efficiency. We compare our work with others in Table 1 and summarize our main contributions below:

Verifiable incremental DPF (VIDPF). First, we introduce a new primitive called verifiable incremental DPF (VIDPF), which builds upon incremental DPFs (IDPF) [16] and verifiable DPFs (VDPF) [32]. VIDPF allows us to verify that clients' inputs are valid by relying on hashing while preserving the client's input privacy.

Batched Consistency Check. Next, we introduce a novel batched consistency check which allows us to drastically reduce the server-to-server communication for the private heavy hitters and make it independent of the total number of clients. At a high level, we succinctly validate the inputs of ℓ clients using a Merkle tree and identify the malformed ones using logarithmic communication. This optimization reduces our server-to-server communication from $\mathcal{O}(\kappa\ell)$ to $\mathcal{O}(\kappa \log_2 \ell)$ bits, where κ is the security parameter, for $\mathcal{O}(1)$ malicious clients. More formally, for ℓ' malicious clients each server communicates $\mathcal{O}(\kappa) \times \min(\ell' \log \ell, (\ell - \ell') \log \frac{\ell}{\ell-\ell'})$ bits.

PLASMA framework. We combine the above two techniques to construct PLASMA, a protocol for private histograms and private heavy hitters in the three-party setting that guarantees full security against a malicious server and malicious clients while maintaining succinct server-to-server communication. PLASMA relies only on efficient hashing and cheap field additions rather than expensive general-purpose MPC or field multiplications. Due to our novel VIDPF primitive, PLASMA outperforms Poplar with regards to runtime by a factor of $3-6\times$. Similarly, our batched consistency check optimization enables us to drastically outperform both Poplar and the sorting-based protocols in terms of server-to-server communication by factors of $182\times$ and $235\times$, respectively. PLASMA is the first work to consider different thresholds for heavy hitters based on pre-agreed prefixes by the servers, allowing for more elaborate private statistics, such as the GPS application discussed later.

Real-world applications. We evaluate PLASMA for two applications of private heavy hitters: one that detects frequently visited URLs and another that identifies popular areas.

- *Popular URLs.* A prominent application (discussed both in [5] and [16]) is identifying which URLs crash the clients' browsers more frequently. In this scenario, each client has a string of n bits that represents the last URL that crashed their browser. In our evaluations (Section 7), we consider $n = 256$ bits, which is sufficient for standard domain names. PLASMA computes the heavy hitter URLs that caused more than $\mathcal{T} = 0.1\%$ of client browsers to crash. We perform the task in 20 minutes for 1 million clients while incurring 200 MB of server-to-server communication.

– *Popular GPS coordinates.* We demonstrate a new application where PLASMA identifies popular areas without sacrificing user privacy. This can help with restaurant recommendations, traffic avoidance, as well as advertising (e.g., businesses can identify crowded shopping areas and target their marketing efforts) while ensuring the GPS coordinates of the users remain private to the servers. Likewise, ride-sharing services can enhance vehicle distribution in busy areas and proactively dispatch more drivers to active areas during rush hour. This is possible by encoding the client GPS coordinates as *plus codes* [47]; PLASMA represents plus codes using 64-bit strings to compute the most popular neighborhoods among a set of client-provided GPS coordinates. We compute the heavy hitter plus codes (i.e., popular coordinates) that more than $\mathcal{T} = 0.1\%$ of clients submitted in less than 3 minutes for 1 million clients.

Table 1: Threat model comparisons, client input validation, and server-to-server communication. All works protect privacy against a malicious server.

Protocol	**Correctness & Privacy Against Malicious Corruption**			**Client Input Validation**	**Succinct Server-to-Server Communication**
	Clients	**Server**	**Server & Clients (Full Security)**		
DPF [17, 18, 37]	●	○	○	○	○
Poplar (IDPF) [16]	●	○	○	●	○
Bucketization (DP) [2]	●	○	○	●	○
MPC-based [15]	○	●[†]	○	○	○
Sorting-based [5, 42]	●	●	●	○	○
PLASMA (this work)	●	●	●	●	●

[†] [15] is using general-purpose MPC and its security relies on the underlying MPC framework. The authors provide both semi-honest and maliciously secure implementations with MP-SPDZ [44] and SCALE-MAMBA [1], respectively.

1.2 Related Work

In this section, we discuss several recent works for private heavy hitters. These works can be classified into four main groups: those based on DPFs, those based on differential privacy (DP), those based on MPC sorting techniques, and finally those based on general-purpose MPC. We summarize the threat models of related works and their server-to-server communication in Table 1.

1.2.1 DPF-based. Distributed point functions [37, 17, 18] offer a straightforward solution for private histograms but they fail for heavy-hitters due to the quadratic blowup in key-size. This was addressed by Poplar [16], which uses two non-colluding servers and introduces the notion of incremental DPFs to allow efficient evaluation of strings based on prefixes. Poplar is robust against malicious clients but is susceptible to additive attacks by a malicious server. To address this, a non-interactive zero-knowledge (NIZK) proof [21, 12, 54, 10, 24] could be implemented to prove the server-side computation without exposing private information; however, this would not be a practical solution. Another possibility is to design customized interactive zero-knowledge (ZK) protocols [36, 49, 52, 50, 51, 58] based on efficient OT/VOLE protocols [45, 56, 23, 22, 60, 30], but that requires designing a custom circuit for proving incremental DPFs, which can be efficiently proven in ZK. In contrast, PLASMA provides full security against both malicious clients and a malicious server. Also, Poplar still leaks some information about the heavy hitter prefixes to the servers as the servers reconstruct the roots of the paths before they prune them. On the other hand, PLASMA performs a secure comparison over the secret shares and either keeps the node with its sub-tree if $\mathcal{T} > count$ or prunes the sub-tree.

1.2.2 DP-based.

There is also a body of work based on local DP and randomized responses to compute the heavy hitters [28, 7, 57, 6, 20, 61]. These techniques only involve a single server collecting data from clients. However, this method introduces a trade-off between utility and privacy, as it leaks some information about the clients' private data to the server. In contrast, other methods that provide stronger privacy guarantees would require at least two not-colluding servers. Notably, secure computation-based solutions can be modified to achieve DP either by using local DP with similar techniques or by adding a smaller amount of noise in MPC and achieving higher data utility while maintaining privacy.

Likewise, bucketization [2] computes approximate statistics on a permuted version of the clients' data combined with dummy data that are sampled as differentially private noise. Bucketization ensures security against malicious clients, but similarly to Poplar, it does not guarantee correctness (only privacy) in the presence of a malicious server. In contrast, PLASMA focuses on exact statistics and is secure against both malicious clients and a malicious server.

1.2.3 Sorting-based.

The recent works of [5, 42] provide new secure sorting algorithms and construct private heavy-hitter protocols based on the sorted data. They provide security against malicious servers and clients in the three-server setting, where one of the servers can be malicious. However, these solutions incur heavy communication overheads by performing a secure sort under MPC. We also demonstrate a $235\times$ improvement in server-to-server communication of PLASMA compared to [5] as shown in Fig. 14. Notably, these protocols do not ensure client input validation and will abort if one of the clients behaves maliciously.[3] This is problematic for realistic applications where multiple untrusted clients are involved. In contrast, PLASMA preemptively detects a malicious client input and discards it from the computation. Finally, PLASMA can be modified to permit different thresholds for heavy-hitters based on pre-agreed prefixes, allowing for more elaborate statistics. This is not possible for sorting-based heavy-hitter protocols.

1.2.4 General MPC-based.

One could use generic honest-majority MPC protocols [35, 27] to compute private heavy hitters, but an efficient representation of the heavy-hitters problem in terms of addition and multiplication gates is not known. In fact, the work by Böhler and Kerschbaum [15] provides a generic MPC-based protocol for computing differentially private heavy hitters. These authors use MPC frameworks like MP-SPDZ [44] and SCALE-MAMBA [1] to achieve semi-honest and malicious security, respectively, but their solution suffers from high communication and slow runtimes.

2 Preliminaries

In this section, we discuss the underlying cryptographic primitives and assumptions used for developing our framework.

2.1 Threat Model

Our threat model assumes three non-colluding servers $(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2)$ that run the histogram/heavy-hitters protocol and ℓ clients. The clients provide inputs to the servers and the servers do not possess any private input. We assume that a central adversary $\mathcal{A}$ maliciously corrupts one of the servers and $\widetilde{\ell} < \ell$ clients.

Clients. Malicious clients may try to deviate from the protocol in order to disproportionally influence the result or even completely corrupt the output of the protocol. PLASMA is robust against malicious clients and preemptively rejects any malformed client input before incorporating it into the computation.

Servers. Similarly, a malicious server may try to deviate from the protocol specification for different reasons. First, a malicious server may attempt to learn private user inputs; PLASMA protects input *privacy* against one malicious server. Another possible attack for a malicious server would be to over-influence or corrupt the

[3] While these protocols can be modified to detect malformed inputs, this requires additional checks per client that rely on expensive field multiplications or use multi-verifier zero-knowledge [3, 59], which results in significant overheads.

result of the protocol. Contrary to some prior works [16, 2], PLASMA protects *correctness* against a malicious server. We say that PLASMA is *robust* against a malicious server, since it protects both correctness and privacy. Hence, PLASMA provides full security against the collusion of one malicious server and malicious clients.

2.2 Notation

We denote the computational security parameter by κ and the statistical security parameter by μ. Let $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{2(\kappa+1)}$ be a pseudorandom generator and $\mathsf{Convert} : \{0,1\}^\kappa \to \mathbb{G}$ be a map converting a random κ-bit string to a pseudorandom group element of $\mathbb{G}$. We use $\coloneqq$ for assignment, $\xleftarrow{R} \mathcal{D}$ for sampling from distribution $\mathcal{D}$, $=$ for checking equality, and $\|$ for concatenation. We define a public set $\mathbf{X}$ with m n-bit strings as $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$ where the ith bit string is denoted as x_i for $i \in [m]$ and the jth bit in $x_i \in \{0,1\}^n$ is denoted as $x_{i,j}$ for $j \in [n]$. We denote the first L bits of x_i as $x_{i,\leq L} \coloneqq (x_{i,1}, x_{i,2}, \ldots x_{i,L})$ for $L \leq n$. Finally, we denote a private n-bit string α and its bit decomposition as $\alpha_1, \ldots, \alpha_n \in \{0,1\}^n$. Let $\mathcal{S}_b$ denote the bth server, for $b \in \{0,1,2\}$; we consider $b+1 \coloneqq (b+1) \mod 3$ and $b+2 \coloneqq (b+2) \mod 3$. Servers do not possess any input. All our protocols assume ℓ clients, each denoted as $\mathcal{C}_i$ for $i \in [\ell]$. Each client $\mathcal{C}_i$ has an n-bit input string $\alpha_i \in \mathbf{X}$, for $i \in [\ell]$.

2.3 Distributed Point Functions (DPF)

Function secret sharing (FSS) [37, 17, 18] enables secret sharing a function into separate keys, where each party's key allows the party to efficiently generate an additive share of the output $f(x)$ on a given input x. DPFs are a special case of FSS where the function f is a point function $f_{\alpha,\beta}(x) \coloneqq \beta$ if $x = \alpha$ and 0, otherwise. A DPF consists of two algorithms Gen and Eval. The Gen algorithm takes as input the function $f_{\alpha,\beta}$ and outputs two keys key_0 and key_1. The Eval algorithm is the evaluation algorithm on an input x such that $\mathsf{Eval}(0, \mathsf{key}_0, x) + \mathsf{Eval}(1, \mathsf{key}_1, x) = \beta$ for $x = \alpha$, and 0 for $x \neq \alpha$. Privacy ensures (α, β) remains hidden from an adversary in possession of one of the keys (but not both).

Efficient implementations for DPFs allow sharing 2^n elements with shares of only $\mathcal{O}(n)$ size [37, 17, 18, 16, 31]. The latest DPF protocols rely on the GGM construction [38] and evaluate a pseudorandom function (PRF) to expand a tree of pseudorandom number generators (PRGs) and output of the PRF tree leaves. Each DPF input has a distinct path through the tree and generates a unique leaf. By secret sharing the initial seeds between two parties, these parties have a zero function for each leaf. This function can be turned into a point function by puncturing a single path in this tree (i.e., input α), where the values at the GGM nodes differ and the leaf value corresponds to β. For all the other inputs the PRG seeds are programmed to be the same. This *correction operation* is designed to fix at most one difference per level and is the backbone for most of the aforementioned works.

Incremental and Verifiable DPF (IDPF and VDPF). The IDPF [16] and VDPF [32] build on standard DPFs to secret share the weights of a tree w.r.t. a single non-zero path. IDPFs perform this task with linear cost in the number of bits n for strings that share common prefixes [16], whereas using standard DPFs this cost would grow to $\mathcal{O}(n^2)$. IDPFs rely on expensive malicious secure sketching checks to ensure that an IDPF key is not malformed. Meanwhile, the work of [32] considers efficient hashing-based verifiable properties to ensure that a DPF (not IDPF) key is well-formed. Moreover, [32] enables a batched verification procedure with communication proportional to the security parameter. However, VDPFs work only for DPF and not IDPF. We present the VDPF algorithms below:

– VDPF.$\mathsf{Gen}(1^\kappa, f_{\alpha,\beta}) \to (\mathsf{key}_0, \mathsf{key}_1)$. Given the security parameter 1^κ and a function f, output keys $\mathsf{key}_0, \mathsf{key}_1$.
– VDPF.$\mathsf{BatchEval}(b, \mathsf{key}_b, \mathbf{X}) \to (\mathbf{Y}_b, \pi_b)$: For $b \in \{0,1\}$, batch verifiable evaluation takes a set $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$, where each $x_i \in \{0,1\}^n$. It outputs $\mathbf{Y}_b \coloneqq \{y_{b,1}, y_{b,2}, \ldots, y_{b,m}\}$ such that $\mathbf{Y}_0 + \mathbf{Y}_1 = f_{\alpha,\beta}(\mathbf{X})$. π_b is a proof that is used to verify the well-formedness of the output.

Privacy ensures that an adversary in possession of one of the keys (but not both) does not obtain any information about the function f. The verifiability property of VDPF ensures that the proofs π_0 and π_1 are same iff they have been generated from valid keys key_0 and key_1 of a point function.

3 Technical Overview

In this section, we recall the histogram and heavy-hitters protocol by Poplar [16]. Then, we describe our histogram protocol for the sake of exposition. Finally, we describe our heavy hitters protocol.

3.1 Histogram Protocol of Poplar

Poplar first considers the problem of computing private subset histograms. The ideal functionality for the histogram can be found in Fig. 6. In the histogram problem, each client holds an n-bit string α and the servers $\mathcal{S}_0$, $\mathcal{S}_1$ have a small set $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$ of m n-bit strings. Each client secret shares its input α using a DPF as $(\mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathsf{DPF.Gen}(1^\kappa, \alpha, 1, \mathbb{G})$. The client sends key_0 to $\mathcal{S}_0$ and key_1 to $\mathcal{S}_1$. Upon receiving the keys, each server $\mathcal{S}_b$ evaluates the DPF on all the strings $x_i \in \mathbf{X}$ and computes the output share $y_b \in \mathbb{F}^m$ by aggregating the evaluated values as $y_b \coloneqq \sum_{x_i \in \mathbf{X}} \mathsf{DPF.Eval}(b, \mathsf{key}_b, x_i)$. The servers perform the same protocol for multiple clients and aggregate the y_b values in an accumulator Y_b. Finally, the servers exchange Y_0 and Y_1 to compute the output histogram as $Y \coloneqq Y_0 + Y_1$. This protocol requires the client to communicate one key to each server and the server-to-server communication is independent of the number of clients since Y_0 and Y_1 are aggregated values. This protocol preserves client privacy.

However, a malicious client can double vote by generating the DPF keys maliciously such that it contains more than one non-zero point or the DPF output at α is greater than 1. To tackle this issue, Poplar introduces a malicious sketching protocol that ensures that the client input is well-formed. The client runs the same protocol twice, once with the actual input vector $v = 0 \ldots 010 \ldots 0$ (where v contains 1 only at the αth position) and once with $\phi v = 0 \ldots 0\phi 0 \ldots 0$ for a random $\phi \xleftarrow{R} \mathbb{F}$. The client also sends some correlated randomness between the two sessions to allow the servers to verify the well-formedness. Upon receiving the DPF keys for the two runs and the correlated randomness, the servers evaluate the DPFs following the previous protocol to obtain $y_b, \widetilde{y_b} \in \mathbb{F}^m$. Then the servers sample random field elements $\mathbf{r} \xleftarrow{R} \mathbb{F}^m$ to perform a random linear combination over y_b and $\widetilde{y_b}$, and utilize the correlated randomness and ϕ to verify the client input's well-formedness. If the client passes the checks, then the client's DPF output y_b is aggregated in Y_b by server b, otherwise, it is ignored. This approach enforces malicious clients to provide correct inputs. It also preserves the client's privacy against a malicious server. However, it allows a malicious server, say $\mathcal{S}_0$, to introduce additive errors (e.g., $\delta \in \mathbb{F}^m$) in $Y_0' \coloneqq Y_0 + \delta$. That way, the output Y of the histogram would be biased by δ as $Y \coloneqq Y_0' + Y_1 = Y_0 + Y_1 + \delta$. The honest server fails to detect such an additive attack, leading to an error in the correctness of the protocol. Moreover, Poplar also requires m field multiplications leading to an $8\times$ computational overhead (for their maliciously secure protocol over their semi-honest counterpart), and requires $\mathcal{O}(\ell)$ server-to-server communication.

3.2 Our Histogram Protocol

We improve upon Poplar's limitations by (1) introducing one additional server, (2) building upon the primitive of verifiable DPF [32] (Section 2.3), and (3) introducing novel consistency checks in the three-party setting. We claim the following benefits over Poplar:

1. Robustness against a collusion of a malicious server and malicious clients,
2. Lightweight consistency checks for malicious behavior (using only symmetric key operations and field additions),
3. Server-to-server communication is independent of the total number of clients.

In fact, our work provides the first maliciously secure protocol whose server-to-server communication is independent of the total number ℓ of clients. Our servers communicate $\mathcal{O}(L)$ hashes for the consistency checks, where $L \coloneqq \min(\widetilde{\ell} \log_2 \ell, \ell)$ and $\widetilde{\ell}$ is the number of corrupt clients. Similar to Poplar, we also ensure client input validation against malicious clients (i.e., honest servers preemptively detect inconsistent client input and discard it). We first discuss our basic histogram protocol (with $\mathcal{O}(\ell)$ server communication) and then we optimize the server-to-server communication.

3.2.1 Basic Protocol We describe our protocol in the three-server setting by demonstrating the necessary modifications as follows:

Robustness against a Malicious Server. The histogram protocol of Poplar is not robust against a malicious server. Hence, we consider a third server $\mathcal{S}_2$ to allow an honest majority to obtain security against one malicious server with improved efficiency. Each client runs three DPF sessions, one between each pair of servers, with independent randomness but the same input α (i.e., the pairwise evaluation of the DPF keys on point α outputs secret shares of one). The client sends the DPF keys for the sessions to the servers and each server obtains two keys. Upon obtaining the DPF keys, each server evaluates the DPF on all input points in $\mathbf{X}$. It is ensured that if the client behaved honestly then at least one of the three sessions will be evaluated honestly since two of the servers are honest. After aggregating all the clients' inputs, the output histogram is reconstructed across the three sessions. If the output is the same between each pair of servers then the servers behaved honestly and that is considered as the output. If the output is inconsistent across a pair of servers then it indicates that one of the servers behaved maliciously (by launching an additive attack) and the honest servers abort - thus providing robustness against the malicious server.

Client Input Validation. The above protocol assumes that the client computes the DPF evaluation keys honestly and sends them to the servers. A malicious client could construct malformed DPF keys such that the client's input gets counted more than once. To prevent such an attack, the work of Poplar considered a consistency check by performing random linear combinations. This led to an $8\times$ overhead in terms of computation as it involved $\mathcal{O}(|\mathbf{X}|)$ multiplications for each client. We avoid such heavy computation and only rely on hashing to perform the consistency check.

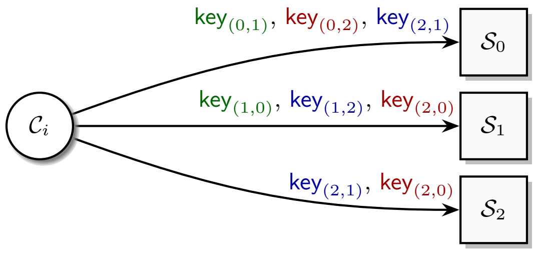
We first ensure that the DPF output is non-zero at a single point. The work of [32] introduces the primitive of verifiable DPF (VDPF). It is a stronger notion of DPF, where the servers obtain proof (of correct evaluation) π upon evaluating a pair of DPF keys on a given input point. The two servers obtain the same proof π if the client generated the DPF keys honestly (i.e., the DPF output is non-zero at a single point α). Multiple proofs corresponding to different evaluation points are batch verified. Next, we ensure that the DPF output value at the non-zero point is indeed 1. Our protocol also instructs the servers to sum up all the output shares (corresponding to each point in $\mathbf{X}$) of the client and reconstruct the output. If the reconstructed output is not 1, then the client's input is discarded since it is not well-formed. If the output is 1, i.e., the client has behaved honestly, then the DPF output shares are aggregated by the server in the output histogram share. This does not quite provide full security against a malicious client since a malicious client could provide inconsistent inputs across the three server sessions.

3.2.2 Final Protocol We further optimize our protocol to reduce server-to-server communication and computation. We also detect a client providing inconsistent input across the three server sessions.

Batched Client Verification. Our basic histogram protocol incurs a server-to-server communication of $\mathcal{O}(\kappa)$ bits for each client to verify the proofs for validating the clients' inputs. This results in $\mathcal{O}(\kappa \times \ell)$ server-to-server communication for ℓ clients.

We propose a technique that allows us to reduce the server-to-server communication. This optimization allows us to batch all the clients' VDPF evaluations using a Merkle tree that has ℓ leaves for ℓ clients. First, the servers succinctly check the equality of ℓ leaves between two servers using hashes. If the roots match then the leaves are the same. If they differ then the servers recursively repeat the same process for each of the two children of the parent node. Proceeding this way, the servers identify the malformed leaves on which the two trees differ. This reduces our server-to-server communication for the consistency check to $\mathcal{O}(\kappa \log_2 \ell)$

bits if $\ell' = \mathcal{O}(1)$ clients behave maliciously. More details can be found in Section 6. For ℓ' malicious clients, each server sends $\mathcal{O}(\kappa) \times \min(\ell' \log \ell, (\ell - \ell') \log \frac{\ell}{\ell - \ell'})$ bits.

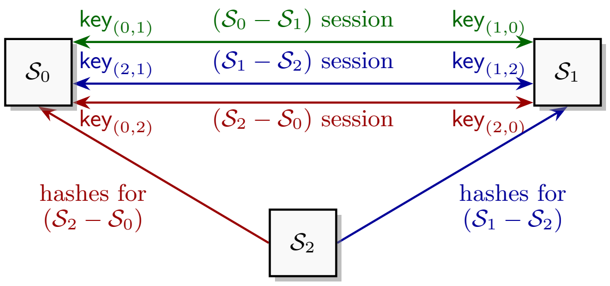


$\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)}$
$\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)}$
$\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)}$
$\mathcal{C}_i$
$\mathcal{S}_0$
$\mathcal{S}_1$
$\mathcal{S}_2$

Fig. 1: Distribution of session keys by client $\mathcal{C}_i$.

Reducing Server-to-Server Latency. We empirically observed that the server-to-server latency increases if there is pairwise communication between the three servers for consistency checks. There are three server-to-server sessions for each client, and the third server $\mathcal{S}_2$ is involved in two of the three sessions: specifically, sessions $\mathcal{S}_1 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_0$. The client generates $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)})$ for session $\mathcal{S}_0 - \mathcal{S}_1$, $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)})$ for session $\mathcal{S}_1 - \mathcal{S}_2$, and $(\mathsf{key}_{(0,2)}, \mathsf{key}_{(2,0)})$ for session $\mathcal{S}_2 - \mathcal{S}_0$. $\mathcal{S}_0$ receives $\mathsf{key}_{(0,1)}$ and $\mathsf{key}_{(0,2)}$ from the client for sessions $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$, respectively. $\mathcal{S}_1$ receives $\mathsf{key}_{(1,0)}$ for session $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathsf{key}_{(1,2)}$ for $\mathcal{S}_1 - \mathcal{S}_2$, while $\mathcal{S}_2$ receives $\mathsf{key}_{(2,1)}$ and $\mathsf{key}_{(2,0)}$ for sessions $\mathcal{S}_1 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_0$, respectively. Additionally, the protocol instructs the client to send $\mathsf{key}_{(2,1)}$ to server $\mathcal{S}_0$ and $\mathsf{key}_{(2,0)}$ to server $\mathcal{S}_1$ respectively. The key distribution process by the client can be visualized in Fig. 1.

This optimization allows $\mathcal{S}_2$ to replicate the computation of $\mathcal{S}_0$ in session $\mathcal{S}_1 - \mathcal{S}_2$ (because they both have $\mathsf{key}_{(2,1)}$) and $\mathcal{S}_2$ acts as an attestator by sending hashes to $\mathcal{S}_1$ of the same messages as $\mathcal{S}_0$ should send. These hashes prevent $\mathcal{S}_0$ from acting maliciously. Similar protocol steps are run by $\mathcal{S}_2$ to attest the $\mathcal{S}_2 - \mathcal{S}_0$ session and prevent $\mathcal{S}_1$ from acting maliciously. We depict this attestation process in Fig. 2. This optimization allows us to batch-verify all three sessions as a single session between $\mathcal{S}_0$ and $\mathcal{S}_1$ using hashes. Combined with the previous optimization (i.e., batched client verification), we can further reduce the communication needed for consistency checks by three times, where each Merkle tree leaf corresponds to three sessions (instead of one). It also allows us to ensure client input consistency across sessions, as described next.



$\mathsf{key}_{(0,1)}$
$(\mathcal{S}_0 - \mathcal{S}_1)$ session
$\mathsf{key}_{(1,0)}$
$\mathsf{key}_{(2,1)}$
$(\mathcal{S}_1 - \mathcal{S}_2)$ session
$\mathsf{key}_{(1,2)}$
$\mathsf{key}_{(0,2)}$
$(\mathcal{S}_2 - \mathcal{S}_0)$ session
$\mathsf{key}_{(2,0)}$
$\mathcal{S}_0$
$\mathcal{S}_1$
$\mathcal{S}_2$
hashes for $(\mathcal{S}_2 - \mathcal{S}_0)$
hashes for $(\mathcal{S}_1 - \mathcal{S}_2)$

Fig. 2: Session keys and attestation by $\mathcal{S}_2$.

Client Input Consistency Across Sessions. A malicious client can provide inconsistent inputs across the three server sessions by providing DPF keys for different points α_1, α_2, and α_3 for three different sessions. The verifiability of the VDPF fails to detect it since each of the individual VDPFs is valid.

To address the issue, we construct a novel consistency check that relies on a single hash verification. Let us denote $\mathbf{Y}_{(0,1)}$, $\mathbf{Y}_{(0,2)}$, and $\mathbf{Y}_{(2,1)}$ be the output of VDPF evaluation by $\mathcal{S}_0$ on keys $\mathsf{key}_{(0,1)}$, $\mathsf{key}_{(0,2)}$, and $\mathsf{key}_{(2,1)}$ corresponding to sessions between $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, and $\mathcal{S}_2 - \mathcal{S}_1$, respectively. Similarly, let us denote $\mathbf{Y}_{(1,0)}$, $\mathbf{Y}_{(2,0)}$, and $\mathbf{Y}_{(1,2)}$ be the output of VDPF evaluation by $\mathcal{S}_1$ on keys $\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(2,0)}$, and $\mathsf{key}_{(1,2)}$ corresponding to sessions between $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, and $\mathcal{S}_2 - \mathcal{S}_1$, respectively. By definition, reconstructing each pair of secret shared outputs (e.g., $\mathbf{Y}_{(0,1)}$, $\mathbf{Y}_{(1,0)}$) results in a vector of zeros except a single location. Note that the client has also sent $\mathsf{key}_{(2,1)}$ to $\mathcal{S}_0$ and $\mathsf{key}_{(2,0)}$ to $\mathcal{S}_1$ respectively. Server $\mathcal{S}_0$ sends hash $h \coloneqq \mathrm{H}(\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} \,\|\, \mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)})$ to $\mathcal{S}_1$, who verifies that $h = \mathrm{H}(\mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)} \,\|\, \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)})$. The verification of the hash h ensures that the client's input is consistent between: (1) the sessions $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_0 - \mathcal{S}_2$, as well as (2) the sessions $\mathcal{S}_0 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_1$. By transitivity, all three sessions are consistent if the hash verification succeeds. Observe that if the servers acted honestly, $\mathbf{Y}_{(0,1)} + \mathbf{Y}_{(1,0)} = \mathbf{Y}_{(0,2)} + \mathbf{Y}_{(2,0)} = \mathbf{Y}_{(1,2)} + \mathbf{Y}_{(2,1)}$ and thus, $\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} = \mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)}$ and $\mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)} = \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)}$. Our novel check requires additions (without any multiplications) and a cheap hash computation. The communication cost is one hash of size κ bits.

We refer to Section 4 for the detailed protocol description and Fig. 7 for the formal protocol details respectively.

3.3 Our Heavy Hitters Protocol via $\mathcal{T}$-Prefix Count Queries

The work of Poplar reduced the problem of computing heavy hitters to the problem of computing prefix count queries for a given prefix $p \in \{0,1\}^*$ over client inputs. Then, they implemented prefix count queries by relying on incremental distributed point functions (IDPF in Section 2.3). However, their protocol leaks the count of strings that contain the $\mathcal{T}$ heavy-hitting prefix p due to the reliance on a prefix-count query oracle that outputs the exact count. To tackle this leakage, we introduce the notion of $\mathcal{T}$-threshold prefix-count queries which returns 1 if at least $\mathcal{T}$ of clients' input strings contain prefix p, otherwise, it returns 0. We define it as follows:

Definition 1 ($\mathcal{T}$-Prefix-count queries). *Return* 1 *(on input prefix $p \in \{0,1\}^*$) if prefix p appears at least $\mathcal{T}$ times in the clients' input strings $\alpha_1, \alpha_2, \ldots, \alpha_\ell \in \{0,1\}^*$ where client $\mathcal{C}_i$ has input string α_i for $i \in [\ell]$, otherwise, return 0.*

Next, we present the high-level idea behind our construction of a $\mathcal{T}$-heavy hitters protocol (for threshold $\mathcal{T}$) given an oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p, \mathcal{T})$ for securely computing $\mathcal{T}$-prefix-count queries over prefix p for the client input strings $\alpha_1, \ldots, \alpha_\ell$.

$\mathcal{T}$-Heavy hitters. The $\mathcal{T}$-Heavy hitters algorithm is provided with oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p, \mathcal{T})$ for computing $\mathcal{T}$-prefix count. The algorithm starts from the empty string ϵ. At each level k, it considers the heavy-hitter prefixes $p \in \{0,1\}^k$ of length in set HH^k, which contains the list of k-bit strings that appear at least k times. The algorithm performs a breadth-first search of the prefix tree. It includes $k+1$ bit length strings $p \,\|\, 0$ in HH^{k+1} if $p \,\|\, 0$ occurs at least $\mathcal{T}$ times in the input strings $(\alpha_1, \ldots, \alpha_\ell)$, otherwise it gets pruned along its subtree. This is performed by querying the oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \,\|\, 0, \mathcal{T})$. The same process is repeated for $p \,\|\, 1$. The algorithm repeats this for all k-bit strings in HH^k (which updates HH^{k+1} based on the search and pruning of set HH^k). At the end of the breadth-first search and the pruning, the algorithm outputs the set of strings that are $\mathcal{T}$-heavy hitters. Our formal algorithm is presented in Fig. 3.
Efficiency. There are ℓ input strings in total. For any string of length k, there are at most ${}^{\ell}/_{\mathcal{T}}$ candidate heavy hitter strings. At each level k, the algorithm makes at most one oracle query per heavy hitter string. Hence, the algorithm makes at most ${}^{n\ell}/_{\mathcal{T}}$ prefix-count-oracle queries for n levels. If we set the threshold to be a constant fraction of all input strings (e.g., $\mathcal{T} = 0.01\ell$), then the number of prefix-count queries are independent of the number of input strings (e.g., ${}^{n\ell}/_{\mathcal{T}} = {}^{n\ell}/_{0.01\ell} = 100n$).

3.4 Implementing $\mathcal{T}$-Prefix Count Queries via Verifiable Incremental DPF

We implement the $\mathcal{T}$-Prefix Count Queries from Def. 1 by relying on a new primitive called verifiable incremental DPF (VIDPF) and invoking the ideal functionality $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 9) for comparison.

$\mathcal{T}$-Heavy Hitters from $\mathcal{T}$-prefix count queries

Parameters: Threshold $\mathcal{T} \in \mathbb{N}$ and string length $n \in \mathbb{N}$.

Inputs: The algorithm has no explicit input. It has access to t-prefix count query oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,t)$ for securely computing t-prefix-count queries over prefix p for strings $\alpha_1,\ldots,\alpha_\ell$.

Outputs: The set of $\mathcal{T}$-heavy-hitter strings in $\alpha_1,\alpha_2,\ldots,\alpha_\ell$.

Algorithm:

- Initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} \coloneqq \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
- For each prefix $p \in \mathsf{HH}^k$ of length k-bits in set HH^k (where $k = 0,1,2,\ldots n-1$) :
  - If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 0\}$.
  - If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 1\}$.
- Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.

Fig. 3: Algorithm for computing $\mathcal{T}$-heavy hitters from $\mathcal{T}$-prefix count queries.

3.4.1 Verifiable Incremental DPF (VIDPF) A DPF allows a client to succinctly share a vector of size 2^n with a single non-zero point. Meanwhile, an incremental DPF (introduced by Poplar and denoted as IDPF) allows the client to succinctly secret share a path in the binary tree (used for representing 2^n leaves in binary format) and each node in the path can hold non-zero values. Our novel VIDPF primitive offers strong integrity guarantees over IDPFs, since the evaluation of the client keys also provides proofs $(\pi_1,\ldots,\pi_n)$ to the servers ensuring that the VIDPF output is non-zero along a single path in the binary tree. It also allows incremental evaluation of the VIDPF over an input $x \in \{0,1\}^k$, given state st_b^{k-1} and proof π_b^{k-1}, corresponding to VIDPF evaluation of the first $k-1$ bits of x. The incremental evaluation enables a evaluator (possessing key_b) to evaluate one level and obtain the secret sharing of output $f(x)$, a new state st_b^k, and a new proof π_b^k corresponding to the VIDPF evaluation of the path involving x. More formally, we capture the high-level ideas of VIDPF using the following two algorithms:

- $\mathsf{Gen}(1^\kappa, 1^n, \alpha, (\beta^1,\beta^2,\ldots,\beta^n), \mathbb{G}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$: Given security parameter κ, input size n, an input string $\alpha \in \{0,1\}^n$ and values $\beta^1,\beta^2,\ldots\beta^n$ the key generation algorithm outputs two VIDPF keys key_0 and key_1.
- $\mathsf{EvalPrefix}(b, \mathsf{key}_b, x \in \{0,1\}^k, \mathsf{st}_b^{k-1}, \pi_b^{k-1}) \rightarrow (\mathsf{st}^k, y_b, \pi_b^k)$: Given a VIDPF key key_b and an input string $x \in \{0,1\}^k$ of length $k \leq n$ bits, the evaluation algorithm outputs an internal state st^k, secret-shared value $y_b \in \mathbb{G}$ and a proof $\pi_b^k \in \{0,1\}^*$.

Correctness of the VIDPF ensures that for all input points $\alpha \in \{0,1\}^n$, output values $\beta^1,\ldots,\beta^n \in \mathbb{G}$, VIDPF keys generated as $(\mathsf{key}_0,\mathsf{key}_1) \leftarrow \mathsf{Gen}(\alpha,\beta^1,\beta^2,\ldots,\beta^n,\mathbb{G})$ and all values $x \in \{0,1\}^k$, where $k \leq n$, the following holds for all $k \leq n$:

$$\pi_0^k = \pi_1^k \text{ and } y = (y_0 + y_1) = \begin{cases} \beta^k, & \text{if } x \text{ is a prefix of } \alpha, \\ 0, & \text{otherwise,} \end{cases}$$

where $(\mathsf{st}_0^k, y_0, \pi_0^k) \coloneqq \mathsf{EvalPrefix}\ (0, \mathsf{key}_0, x, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$ and $(\mathsf{st}_1^k, y_1, \pi_1^k) \coloneqq \mathsf{EvalPrefix}\ (1, \mathsf{key}_1, x, \mathsf{st}_1^{k-1}, \pi_1^{k-1})$. For security guarantees, we require two additional properties from the VIDPF primitive:

- *Input Privacy.* The security of VIDPF guarantees that an adversarial evaluator in possession of either key_0 or key_1 (but not both), does not learn any information about either the input α or the outputs $\beta^1,\ldots,\beta^n$ of the client.
- *Verifiability.* The verifiability property states that if two proofs (e.g., π_0^k and π_1^k) are the same, then there is at most one path of length k in the binary tree whose evaluation with keys $(\mathsf{key}_0,\mathsf{key}_1)$ outputs $(\beta^1,\beta^2,\ldots,\beta^k)$. More formally, for any $k \in [1,\ldots n]$ there exists a single k-bit string $\widetilde{x} \in \{0,1\}^k$ such that if $\pi_0^k = \pi_1^k$, then the following holds:

$$\begin{aligned}&\mathsf{EvalPrefix}(0, \mathsf{key}_0, z, \mathsf{st}_0^{k-1}, \pi_0^{k-1}) + \\ &\mathsf{EvalPrefix}(1, \mathsf{key}_1, z, \mathsf{st}_1^{k-1}, \pi_1^{k-1}) \\ &= \begin{cases} \beta^k, \text{if } z = \widetilde{x}, \\ 0, \text{if } z = \{0,1\}^k \setminus \{\widetilde{x}\}, \end{cases}\end{aligned}$$

where $\mathsf{st}_0^{k-1}, \pi_0^{k-1}$ and $\mathsf{st}_1^{k-1}, \pi_1^{k-1}$ are obtained by running the EvalPrefix algorithm on $k-1$ bits of z respectively. The evaluators initialize $\mathsf{st}_0^0 := \mathsf{st}_1^0 := 0$ and $\pi_0^0 := \pi_1^0 := 0$.

We provide a construction of VIDPF in Figs. 4 and 5 based on length doubling PRG in the random oracle model.

Primitives: $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{2\kappa+2}$ is a pseudorandom generator. $\mathrm{H}_1 : \{0,1\}^* \times \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$ and $\mathrm{H}_2 : \{0,1\}^{2\kappa} \to \{0,1\}^{2\kappa}$ are random oracles.

```
Gen(1^κ, 1^n, α, (β_1, β_2, ... β_n), 𝔾):                                  ▷ Generate DPF keys.
1:  Sample $s_b^{(0)} \xleftarrow{R} \{0,1\}^\kappa$ for $b \in \{0,1\}$            ▷ Secret seeds.
2:  Let $t_0^{(0)} := 0$ and $t_1^{(0)} := 1$
3:  for $i := 1$ to $n$ do                                                 ▷ For each bit of $\alpha$.
4:      $s_b^L \| t_b^L \| s_b^R \| t_b^R := \mathsf{PRG}(s_b^{(i-1)})$ for $b \in \{0,1\}$   ▷ Parse the output of PRG as a sequence of $(\kappa \| 1 \| \kappa \| 1)$ bits.
5:      if $\alpha_i = 0$ then Diff := $L$, Same := $R$                     ▷ Set right children to be equal.
6:      else Diff := $R$, Same := $L$                                      ▷ Set left children to be equal.
7:      $s_{\mathsf{cw}} := s_0^{\mathsf{Same}} \oplus s_1^{\mathsf{Same}}$
8:      $t_{\mathsf{cw}}^L := t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$           ▷ Left control bits not equal if $\alpha_i = 0$.
9:      $t_{\mathsf{cw}}^R := t_0^R \oplus t_1^R \oplus \alpha_i$                  ▷ Right control bits not equal if $\alpha_i = 1$.
10:     $\tilde{s}_b^{(i)} := s_b^{\mathsf{Diff}} \oplus t_b^{(i-1)} \cdot s_{\mathsf{cw}}$ for $b \in \{0,1\}$   ▷ Correction.
11:     $t_b^{(i)} := t_b^{\mathsf{Diff}} \oplus t_b^{(i-1)} \cdot t_{\mathsf{cw}}^{\mathsf{Diff}}$ for $b \in \{0,1\}$   ▷ Correction.
12:     $s_b^{(i)} \| W_b^{(i)} := \mathsf{Convert}(\tilde{s}_b^{(i)})$ for $b \in \{0,1\}$
13:     $W_{\mathsf{cw}}^{(i)} := (-1)^{t_1^{(i)}} \cdot [\beta_i - W_0^{(i)} + W_1^{(i)}]$     ▷ Output correction.
14:     $\mathsf{cw}^{(i)} := s_{\mathsf{cw}} \| t_{\mathsf{cw}}^L \| t_{\mathsf{cw}}^R \| W_{\mathsf{cw}}^{(i)}$     ▷ Correction word for level $i$.
15:     $\widetilde{\pi}_b^{(i)} = \mathrm{H}_1(\alpha_{\le i} \| s_b^{(i)})$
16:     $\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$.
17: $\mathsf{key}_b := (s_b^{(0)} \| \mathsf{cw}^{(1)} \| \ldots \| \mathsf{cw}^{(n)} \| \mathsf{cs}^{(1)} \| \ldots \| \mathsf{cs}^{(n)})$ for $b \in \{0,1\}$   ▷ Key for party $b$.
18: return $\mathsf{key}_b$ for $b \in \{0,1\}$
```

Fig. 4: Protocol π_{VIDPF} for Verifiable incremental DPF (continues in Fig. 5).

Next, we outline our protocol for securely implementing $\mathcal{T}$-prefix count queries using VIDPF and the comparison functionality $\mathcal{F}_{\mathsf{CMP}}$.

3.4.2 Implementing $\mathcal{T}$-Prefix Count Queries

Each client generates three pairs of VIDPF session keys, one for each pair of servers, with independent randomness but the same input point α and output values $(1, 1, \ldots, 1)$. The client sends the VIDPF keys for the sessions to the respective participating servers similarly to our histogram protocol, as indicated in Fig. 1.

Basic Protocol. As depicted in Fig. 2, $\mathcal{S}_2$ behaves as an attestator for the $\mathcal{S}_0 - \mathcal{S}_2$ session and sends hashes of the messages that $\mathcal{S}_1$ should send. The hash prevents server $\mathcal{S}_1$ from acting maliciously corresponding to the $\mathcal{S}_0 - \mathcal{S}_2$ session. Similar protocol steps are run by $\mathcal{S}_2$ for the session between $\mathcal{S}_1 - \mathcal{S}_2$, where $\mathcal{S}_2$ sends hashes to $\mathcal{S}_1$. Hence, $\mathcal{S}_0$ and $\mathcal{S}_1$ run three sessions, and $\mathcal{S}_2$ runs two of those sessions in parallel. Next, we describe the protocol to compute a $\mathcal{T}$-prefix count query on a string $p \,\|\, 0 \in \{0,1\}^k$ (note, the same process can be repeated for query string $p \,\|\, 1$). The servers $\mathcal{S}_0$ and $\mathcal{S}_1$ evaluate the VIDPF keys for the three sessions on $p \,\|\, 0$ and obtain a secret share of the output $y^{p\|0}$ and proof π. Ideally, $y^{p\|0}$ should be $\beta^k = 1$ for an honest

```
$\mathsf{EvalNext}(b, i, \mathsf{st}^{(i-1)}, \mathsf{cw}^{(i)}, \mathsf{cs}^{(i)}, x_{\leq i}, \pi)$:    ▷ Evaluate $x_i$.
1: Parse $\mathsf{st}^{(i-1)}$ as $(s^{i-1} \parallel t^{i-1})$.
2: $s_{\mathsf{cw}} \parallel t^L_{\mathsf{cw}} \parallel t^R_{\mathsf{cw}} \parallel W^{(i)}_{\mathsf{cw}} := \mathsf{cw}^i$    ▷ Parse correction word.
3: $\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R := \mathsf{PRG}(s^{(i-1)})$    ▷ Parse the output of PRG as a sequence of $(\kappa \parallel 1 \parallel \kappa \parallel 1)$ bits.
4: $\tau^{(i)} := (\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R) \oplus (t^{(i-1)} \cdot [s_{\mathsf{cw}} \parallel t^L_{\mathsf{cw}} \parallel s_{\mathsf{cw}} \parallel t^R_{\mathsf{cw}}])$
5: $s^L \parallel t^L \parallel s^R \parallel t^R := \tau^{(i)}$    ▷ Parse $\tau^{(i)}$.
6: if $x_i = 0$ then $\tilde{s}^{(i)} := s^L, \quad t^{(i)} := t^L$    ▷ Keep left path.
7: else $\tilde{s}^{(i)} := s^R, \quad t^{(i)} := t^R$    ▷ Keep right path.
8: $s^{(i)} \parallel W^{(i)} := \mathsf{Convert}(\tilde{s}^{(i)})$    ▷ New seed and output for level $i$.
9: $\mathsf{st}^{(i)} := s^{(i)} \parallel t^{(i)}$    ▷ Save the state.
10: $y^{(i)} := (-1)^b \cdot [W^{(i)} + t^{(i)} \cdot W_{\mathsf{cw}}]$    ▷ Compute output at level $i$.
11: $\widetilde{\pi}^{(i)} = \mathrm{H}_1(x^{\leq i} \parallel s^{(i)})$.
12: $\pi = \pi \oplus \mathrm{H}_2(\pi \oplus (1 - t^{(i)}) \cdot \widetilde{\pi}^{(i)} \oplus t^{(i)} \cdot \mathsf{cs}^{(i)})$.
13: return $(\mathsf{st}^{(i)}, y^{(i)}, \pi)$

$\mathsf{EvalPrefix}(b, \mathsf{key}, x \in \{0,1\}^n, \mathsf{st}^{(d-1)}, d, \pi)$:    ▷ Evaluate one public bitstring $x$ on all it's bits $x_i$ for $i \in [n]$.
1: Parse key as $s^{(0)} \parallel \mathsf{cw}^{(1)} \parallel \ldots \parallel \mathsf{cw}^{(n)} \parallel \mathsf{cs}^{(1)} \parallel \ldots \parallel \mathsf{cs}^{(n)}$.    ▷ Parse key for party $b$.
2: if $(d \neq 1)$ then parse $\mathsf{st}^{(d-1)}$ as $(s^{(d-1)} \parallel t^{(d-1)})$,
3: else $t^{(0)} := b, \quad \mathsf{st}^{(0)} := s^{(0)} \parallel t^{(0)}$.
4: for $i := d$ to $n$ do    ▷ For each bit of $x$.
5:     $(\mathsf{st}^{(i)}, y^{(i)}, \pi) := \mathsf{EvalNext}(b, i, \mathsf{st}^{(i-1)}, \mathsf{cw}^i, x^{\leq i}, \pi)$.
6: return $(\mathsf{st}^{(n)}, y^{(n)}, \pi)$
```

Fig. 5: Protocol π_{VIDPF} for Verifiable incremental DPF (continuing from Fig. 4).

client. However, a malicious client could construct malformed VIDPF keys such that the client's input gets counted more than once.

Client Input Validation. We introduce the following consistency checks to validate a client's input:

1. The servers $\mathcal{S}_0$ and $\mathcal{S}_1$ first verify that the proofs π are the same for all three sessions. This ensures that there is at most one path in the binary tree that is non-zero.
2. Next, for the root level (i.e., $k = 0$), the servers evaluate the VIDPF keys on the empty string ϵ and verify that it is 1.
3. Finally, the servers need to verify that $y^{p\|0}$ is either 0 or 1, without reconstructing the output. This is ensured by performing a novel consistency check on the subtree involving $p \parallel 0$. The servers evaluate the VIDPF keys on the parent string p and sibling (of $p \parallel 0$) string $p \parallel 1$ to obtain secret shares of the output of y^p and $y^{p\|1}$ respectively. The servers reconstruct $y^p - (y^{p\|0} + y^{p\|1})$ and verify that it is 0 (at most one child can equal 1 when a parent holds a value of 1). This ensures that the subtree involving $p \parallel 0$ is valid. This step is repeated iteratively for the path (in the subtree) involving p, until all layers are processed. Combining all k checks ensures that $y^{p\|0} = 1$ iff $y^p = 1$ and $y^{p\|1} = 0$, else $y^{p\|0} = 0$. The servers also verify the corresponding proofs π generated during the VIDPF evaluation along the path to ensure there is at most one path in the entire binary tree that is non-zero.
4. The servers also need to ensure that the client input is consistent across the three server sessions. This is ensured by computing the difference of the reconstructed outputs across the sessions and verifying that they are equal to 0 by matching their hash values.

Output Phase. Once the client's VIDPF output $y^{p\|0}$ is verified, the secret shares of $y^{p\|0}$ are aggregated into $\mathsf{count}^{y\|0}$. The servers repeat the above steps for all the clients in parallel to obtain secret shares of $y^{p\|0}$. The servers invoke the comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 9) with the secret shares of count and threshold $\mathcal{T}$. $\mathcal{F}_{\mathsf{CMP}}$ reconstructs count and it outputs 1 if $\mathsf{count} \geq \mathcal{T}$, otherwise, it outputs 0. This is returned by the servers as the output of the $\mathcal{T}$-prefix count oracle query response to the string $y \parallel 0$. The comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [48].

Robustness Against a Malicious Server. The third server ensures that if the client behaved honestly then at least one of the three sessions will be evaluated correctly since two of the servers are honest. After aggregating all the client's inputs, count is reconstructed across the three sessions by $\mathcal{F}_{\mathsf{CMP}}$. If count is inconsistent across any pair of servers then $\mathcal{F}_{\mathsf{CMP}}$ returns $\perp$ indicating that one of the servers behaved maliciously by launching an additive attack. This will cause the honest servers to abort, providing robustness against the malicious server.

In our final protocol, we verify multiple client inputs at each level in one batch, similar to our histogram protocol. This ensures that our server-to-server communication is independent of the total number of clients, and depends only on the number of malicious clients and the number of heavy-hitter prefix strings. We refer the reader to Section 5 for the protocol description and Figs. 10 and 11 for the formal protocol details. We also present our heavy-hitters protocol for different thresholds in Appendix A.

4 Private Histogram

We provide the ideal functionality $\mathcal{F}_{\mathsf{HIST}}$ for histogram between three servers and ℓ clients in Fig. 6. An adversary $\mathcal{A}$ can maliciously corrupt any one of the servers and multiple clients.

Functionality $\mathcal{F}_{\mathsf{HIST}}$

Parameters: Servers $\mathcal{S}_0, \mathcal{S}_1$ and $\mathcal{S}_2$, and ℓ clients $\mathcal{C}_i$ for $i \in [\ell]$. Servers $\mathcal{S}_0$, $\mathcal{S}_1$ and $\mathcal{S}_2$ agree upon:

– $\mathbf{X}$ to be a public set of m n-bit strings $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$.
– A bound ℓ on the number of client submissions.

Inputs:

– *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$: No input.
– *Clients* $\mathcal{C}_i$: A point $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$.

Outputs: Initialize $\mathsf{HIST} \coloneqq 0^m$. Compute histogram $\mathsf{HIST} \coloneqq \{y_1, \ldots, y_m\}$ based on client inputs α_i for $i \in [\ell]$ and $j \in [m]$:

$$y_j \coloneqq \begin{cases} y_j + 1, & \text{if } \alpha_i = x_j, \\ y_j, & \text{otherwise.} \end{cases}$$

$\mathcal{F}_{\mathsf{HIST}}$ outputs the following:

– *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$: Histogram HIST
– *Clients* $\mathcal{C}_i$: No output for $i \in [\ell]$

Corruption: Adversary $\mathcal{A}$ maliciously corrupts one server and multiple clients together. If $\mathcal{A}$ instructs the functionality to abort by sending $\perp$, the functionality instructs the honest servers to output $\perp$.

Fig. 6: The ideal $\mathcal{F}_{\mathsf{HIST}}$ functionality for histogram.

Our detailed protocol π_{HIST} that implements $\mathcal{F}_{\mathsf{HIST}}$ appears in Fig. 7, and high-level ideas of our protocol can be found in Section 3.2. Our π_{HIST} protocol computes a private histogram that represents the data distribution of ℓ clients while protecting the privacy of the individual data points. π_{HIST} runs on three servers $(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2)$ that utilize the verifiable DPF (VDPF) protocol [32] to privately aggregate the clients' data points. Additionally, π_{HIST} runs three VDPF sessions, which guarantees security against a malicious server. We denote the secret shared variables of a session between $\mathcal{S}_{b_1}$ and $\mathcal{S}_{b_2}$ with a subscript of (b_1, b_2) if they are local to $\mathcal{S}_{b_1}$ and with a subscript (b_2, b_1) if they are local to $\mathcal{S}_{b_2}$. Our protocol proceeds in three phases: a client computation phase, a server computation phase, and an output phase.

Client Computation. During the client computation phase, each client $\mathcal{C}$ prepares three pairs of VDPF keys for their private data point $\alpha \in \mathbf{X}$ and output value 1 using independent randomness for each key generation. Using three pairs of keys essentially allows us to run three separate VDPF sessions. $\mathcal{S}_0$ and $\mathcal{S}_1$ each have one key for each of the three sessions, while $\mathcal{S}_2$ acts as a consistency checking server and shares one key with each of the other two servers. More specifically, the client generates $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)})$

for $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)})$ for $\mathcal{S}_1$, and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ for $\mathcal{S}_2$. The client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$, and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$ as shown in Fig. 1.

Server Computation. Each server first initializes one histogram for each session (e.g., $\mathsf{HIST}_{(b_1,b_2)}$ for $b_1, b_2 \in \{0,1,2\}$ and $b_1 \neq b_2$) with m zeros (where $|\mathbf{X}| = m$). The servers start accepting VDPF keys from the clients and perform the following:

(a) Each of the servers evaluates each client submission on all m data points in $\mathbf{X}$ and computes a secret shared vector $\mathbf{Y}_{(b_1,b_2)}$ and a hash $\pi_{(b_1,b_2)}$ that is used for consistency checking by relying on the integrity guarantees of VDPF [32]. By the VDPF construction, $\mathbf{Y}_{(b_1,b_2)}$ is a vector of additive shares of zeros in $m-1$ positions and one share of one in a single position indicated by the client's input a_i. We denote each index j for these shares as $y_{(b_1,b_2),j}$ where b_1 and b_2 indicate the two servers $\mathcal{S}_{b_1}$ and $\mathcal{S}_{b_2}$ that run each session. Each server $\mathcal{S}_{b_1}$ then computes $\tau_{(b_1,b_2)}$ by locally adding the $y_{(b_1,b_2),j}$ values for all $j \in [m]$. $\mathcal{S}_0$ and $\mathcal{S}_1$ communicate the $\tau_{(b_1,b_2)}$ and $\pi_{(b_1,b_2)}$ values for all sessions between them. In particular, $\mathcal{S}_0$ sends $(\tau_{(0,1)}, \pi_{(0,1)}, \tau_{(0,2)}, \pi_{(0,2)}, \tau_{(2,1)}, \pi_{(2,1)})$ to $\mathcal{S}_1$ and $\mathcal{S}_1$ sends $(\tau_{(1,2)}, \pi_{(1,2)}, \tau_{(1,0)}, \pi_{(1,0)}, \tau_{(2,0)}, \pi_{(2,0)})$ to $\mathcal{S}_0$. $\mathcal{S}_2$ sends $\mathrm{H}(\tau_{(2,0)}, \pi_{(2,0)})$ to $\mathcal{S}_0$, and $\mathrm{H}(\tau_{(2,1)}, \pi_{(2,1)})$ to $\mathcal{S}_1$. Finally, $\mathcal{S}_0$ computes a hash $h :- \mathrm{H}(\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} \parallel \mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)})$ and sends it to $\mathcal{S}_1$.

(b) $\mathcal{S}_0$ and $\mathcal{S}_1$ can now locally check that, within each session, all the π values match and that all the τ values add up to 1. This guarantees that there were no additive attacks introduced. Finally, $\mathcal{S}_1$ verifies the clients' input consistency by checking that the value h received from $\mathcal{S}_0$ is equal to $\mathrm{H}(\mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)} \parallel \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)})$. Each server can then aggregate $\mathbf{Y}_{(b_1,b_2)}$ in their histogram $\mathsf{HIST}_{(b_1,b_2)}$, which results in increasing only the bucket that the client's private data point a_i indicates.

Output Phase. After all ℓ client submissions have been processed, each two servers $\mathcal{S}_b$ and $\mathcal{S}_{b+1}$ (for $b \in \{0,1,2\}$) exchange the secret shared values $(\mathsf{HIST}_{(b,b+1)}$ and $\mathsf{HIST}_{(b+1,b)})$ and reconstruct the final histograms as $\mathsf{HIST}_b := \mathsf{HIST}_{(b,b+1)} + \mathsf{HIST}_{(b+1,b)}$. Both $\mathcal{S}_0$ and $\mathcal{S}_1$ verify that $\mathsf{HIST} = \mathsf{HIST}_0 = \mathsf{HIST}_1 = \mathsf{HIST}_2$. In the formal protocol, the servers first send commitments to their shares via hashes and then open the shares. This is required for the formal simulation-based security proof against a rushing adversarial server. In addition, $\mathcal{S}_2$ also attests to the computation corresponding to the two sessions (involving $\mathcal{S}_2$) by sending hashes.

This completes the description of our protocol π_{HIST} (Fig. 7). The security of our protocol is captured in Theorem 1. Formal protocol details can be found below.

Theorem 1. *Assuming VDPF is a verifiable DPF and* H *is a random oracle then* π_{HIST} *(Fig. 7) implements the* $\mathcal{F}_{\mathsf{HIST}}$ *functionality in the random oracle model against malicious corruption of one server and corruption of* $\widetilde{\ell} \leq \ell$ *clients.*

Proof Sketch. The adversary is allowed to corrupt $\ell' \leq \ell$ clients and one of the servers. The other two servers remain uncorrupted. We discuss the ways a malicious client can attempt to inject an error and we demonstrate our consistency checks for each of the cases.

- *Client VDPF keys are malformed.* A malicious client can attempt to provide malformed VDPF keys which are non-zero in more than one leaf in the binary tree (of 2^n leaves). This gets detected in the session involving the honest servers due to the verifiable property of the VDPF when the servers verify the proofs generated during the VDPF evaluation. If the checks pass, then it is ensured that the VDPF keys provided by the client are valid.
- *Client VDPF input is malformed.* Next, a malicious client can try to double-vote on its input point α by constructing the VDPF on $(\alpha, \widetilde{\beta})$. i.e., $f(\alpha) = \widetilde{\beta}$, where $\widetilde{\beta} > 1$, instead of $(\alpha, 1)$. This is detected by the honest servers since the honest servers reconstruct the sum of all the VDPF evaluations over the m input points. This is performed in parallel for all three sessions. The servers verify that the reconstructed output sum is 1 to ensure that the client has voted only once on its secret input point.
- *VDPF input is inconsistent across sessions.* Finally, a malicious client can try to provide different VDPF keys in different sessions, for example it constructs VDPF keys for input $(\alpha_1, 1)$ for the $\mathcal{S}_0 - \mathcal{S}_1$ session, $(\alpha_2, 1)$ for the $\mathcal{S}_1 - \mathcal{S}_2$ session, and $(\alpha_3, 1)$ for the $\mathcal{S}_2 - \mathcal{S}_0$ session, where $\alpha_1 \neq \alpha_2 \neq \alpha_3$. The above two checks would still pass since they ensure client input validation within each session but not client input

Private Histogram π_{HIST}

We denote a vector $\mathbf{Y} \in \mathbb{F}^m$ component-wise as $\mathbf{Y} \coloneqq \{y_1, y_2, \ldots, y_m\}$, where $y_j \in \mathbb{F}$ for $j \in [m]$.

- **Input:** Each client $\mathcal{C}_i$ has an input point $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$.
- **Output:** $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2$ output a histogram of the ℓ clients' data. If the servers abort then it denotes a malicious server involvement.
- **Primitive:** VDPF $\coloneqq$ ($\mathsf{Gen}, \mathsf{BatchEval}$) is a verifiable distributed point function. H is a random oracle.

1: **Client $\mathcal{C}$ Computation.** **(Repeated for ℓ clients, each of which has their own private input α.)**

(a) Client $\mathcal{C}$ with input α prepares three pairs of DPF keys with independent randomness $u, v, w \xleftarrow{R} \{0,1\}^\kappa$, as follows:

$$(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) \coloneqq \mathsf{Gen}(1^\kappa, \alpha, 1, \mathbb{G}), \quad (\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) \coloneqq \mathsf{Gen}(1^\kappa, \alpha, 1, \mathbb{G}), \quad (\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) \coloneqq \mathsf{Gen}(1^\kappa, \alpha, 1, \mathbb{G})$$

(b) The client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$.

2: **Server Computation.** **(Repeated for ℓ clients, each of which has sent their own keys.)**
If this is the first client, each server $\mathcal{S}_b$ initializes $\mathsf{HIST}_{(b,b+1)}$ and $\mathsf{HIST}_{(b+1,b)}$ for $b \in \{0,1,2\}$ as follows:

$$\mathcal{S}_0 \text{ initializes } \mathsf{HIST}_{(0,1)} \coloneqq 0^m, \mathsf{HIST}_{(0,2)} \coloneqq 0^m, \text{ and } \mathsf{HIST}_{(2,1)} \coloneqq 0^m$$

$$\mathcal{S}_1 \text{ initializes } \mathsf{HIST}_{(1,2)} \coloneqq 0^m, \mathsf{HIST}_{(1,0)} \coloneqq 0^m, \text{ and } \mathsf{HIST}_{(2,0)} \coloneqq 0^m, \quad \mathcal{S}_2 \text{ initializes } \mathsf{HIST}_{(2,0)} \coloneqq 0^m \text{ and } \mathsf{HIST}_{(2,1)} \coloneqq 0^m$$

(a) Each server $\mathcal{S}_b$ computes $\mathbf{Y}_{(b,b+1)}$ and $\mathbf{Y}_{(b,b+2)}$ for $b \in \{0,1,2\}$ as follows:

$$\mathcal{S}_0 \text{ computes } \mathbf{Y}_{(0,1)}, \pi_{(0,1)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(0, \mathsf{key}_{(0,1)}, \mathbf{X}) \text{ and } \mathbf{Y}_{(0,2)}, \pi_{(0,2)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(1, \mathsf{key}_{(0,2)}, \mathbf{X})$$

$$\mathcal{S}_1 \text{ computes } \mathbf{Y}_{(1,2)}, \pi_{(1,2)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(0, \mathsf{key}_{(1,2)}, \mathbf{X}) \text{ and } \mathbf{Y}_{(1,0)}, \pi_{(1,0)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(1, \mathsf{key}_{(1,0)}, \mathbf{X})$$

$$\mathcal{S}_0 \text{ and } \mathcal{S}_2 \text{ compute } \mathbf{Y}_{(2,1)}, \pi_{(2,1)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(1, \mathsf{key}_{(2,1)}, \mathbf{X})$$

$$\mathcal{S}_1 \text{ and } \mathcal{S}_2 \text{ compute } \mathbf{Y}_{(2,0)}, \pi_{(2,0)} \coloneqq \text{VDPF}.\mathsf{BatchEval}(0, \mathsf{key}_{(2,0)}, \mathbf{X})$$

Each server $\mathcal{S}_b$ computes $\tau_{(b,b+1)}$ and $\tau_{(b,b+2)}$ for $b \in \{0,1,2\}$ as follows:

$$\mathcal{S}_0 \text{ parses } \mathbf{Y}_{(0,1)} = \{y_{(0,1),1}, y_{(0,1),2}, \ldots, y_{(0,1),m}\} \text{ and computes } \tau_{(0,1)} \coloneqq \textstyle\sum_{j=1}^m y_{(0,1),j}$$

$$\mathcal{S}_0 \text{ parses } \mathbf{Y}_{(0,2)} = \{y_{(0,2),1}, y_{(0,2),2}, \ldots, y_{(0,2),m}\} \text{ and computes } \tau_{(0,2)} \coloneqq \textstyle\sum_{j=1}^m y_{(0,2),j}$$

$$\mathcal{S}_1 \text{ parses } \mathbf{Y}_{(1,2)} = \{y_{(1,2),1}, y_{(1,2),2}, \ldots, y_{(1,2),m}\} \text{ and computes } \tau_{(1,2)} \coloneqq \textstyle\sum_{j=1}^m y_{(1,2),j}$$

$$\mathcal{S}_1 \text{ parses } \mathbf{Y}_{(1,0)} = \{y_{(1,0),1}, y_{(1,0),2}, \ldots, y_{(1,0),m}\} \text{ and computes } \tau_{(1,0)} \coloneqq \textstyle\sum_{j=1}^m y_{(1,0),j}$$

$$\mathcal{S}_1 \text{ and } \mathcal{S}_2 \text{ parse } \mathbf{Y}_{(2,0)} = \{y_{(2,0),1}, y_{(2,0),2}, \ldots, y_{(2,0),m}\} \text{ and compute } \tau_{(2,0)} \coloneqq \textstyle\sum_{j=1}^m y_{(2,0),j}$$

$$\mathcal{S}_0 \text{ and } \mathcal{S}_2 \text{ parse } \mathbf{Y}_{(2,1)} = \{y_{(2,1),1}, y_{(2,1),2}, \ldots, y_{(2,1),m}\} \text{ and compute } \tau_{(2,1)} \coloneqq \textstyle\sum_{j=1}^m y_{(2,1),j}$$

$\mathcal{S}_0$ sends $(\tau_{(0,1)}, \pi_{(0,1)}, \tau_{(0,2)}, \pi_{(0,2)}, \tau_{(2,1)}, \pi_{(2,1)})$ to $\mathcal{S}_1$. $\mathcal{S}_1$ sends $(\tau_{(1,2)}, \pi_{(1,2)}, \tau_{(1,0)}, \pi_{(1,0)}, \tau_{(2,0)}, \pi_{(2,0)})$ to $\mathcal{S}_0$. $\mathcal{S}_2$ sends $\mathrm{H}(\tau_{(2,0)}, \pi_{(2,0)})$ to $\mathcal{S}_0$, and $\mathrm{H}(\tau_{(2,1)}, \pi_{(2,1)})$ to $\mathcal{S}_1$. $\mathcal{S}_0$ also sends hash h to $\mathcal{S}_1$, where h is of the form $h = \mathrm{H}(\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} \parallel \mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)})$.

(b) Each server $\mathcal{S}_b$ sets $\mathsf{ver}_b \coloneqq 1$ for $b \in \{0,1\}$. Then, the servers locally perform the following computation:

$$\mathcal{S}_0 \text{ and } \mathcal{S}_1 \text{ sets } \mathsf{ver}_0, \mathsf{ver}_1 \coloneqq 0 \text{ if } (\pi_{(0,1)} \neq \pi_{(1,0)}) \vee (\pi_{(0,2)} \neq \pi_{(2,0)}) \vee (\pi_{(2,1)} \neq \pi_{(1,2)}) \vee$$

$$(\tau_{(0,1)} + \tau_{(1,0)} \neq 1) \vee (\tau_{(0,2)} + \tau_{(2,0)} \neq 1) \vee (\tau_{(2,1)} + \tau_{(1,2)} \neq 1)$$

$$\mathcal{S}_1 \text{ sets } \mathsf{ver}_1 \coloneqq 0 \text{ if } h \neq \mathrm{H}(\mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)} \parallel \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)})$$

$\mathcal{S}_0$ and $\mathcal{S}_1$ broadcast ver_0 and ver_1. Ignore the client's input if either is 0. Else, aggregate the client input into the histogram as follows:

$$\mathcal{S}_0 \text{ updates } \mathsf{HIST}_{(0,1)} \coloneqq \mathsf{HIST}_{(0,1)} + \mathbf{Y}_{(0,1)}, \mathsf{HIST}_{(0,2)} \coloneqq \mathsf{HIST}_{(0,2)} + \mathbf{Y}_{(0,2)} \text{ and } \mathsf{HIST}_{(2,1)} \coloneqq \mathsf{HIST}_{(2,1)} + \mathbf{Y}_{(2,1)}$$

$$\mathcal{S}_1 \text{ updates } \mathsf{HIST}_{(1,2)} \coloneqq \mathsf{HIST}_{(1,2)} + \mathbf{Y}_{(1,2)}, \mathsf{HIST}_{(1,0)} \coloneqq \mathsf{HIST}_{(1,0)} + \mathbf{Y}_{(1,0)} \text{ and } \mathsf{HIST}_{(2,0)} \coloneqq \mathsf{HIST}_{(2,0)} + \mathbf{Y}_{(2,0)}$$

$$\mathcal{S}_2 \text{ updates } \mathsf{HIST}_{(2,0)} \coloneqq \mathsf{HIST}_{(2,0)} + \mathbf{Y}_{(2,0)} \text{ and } \mathsf{HIST}_{(2,1)} \coloneqq \mathsf{HIST}_{(2,1)} + \mathbf{Y}_{(2,1)}$$

3: **Output Phase**.

(a) Each two servers $\mathcal{S}_b$ and $\mathcal{S}_{b+1}$ exchange $\mathrm{H}(\mathsf{HIST}_{(b,b+1)}, r_{(b,b+1)})$ and $\mathrm{H}(\mathsf{HIST}_{(b+1,b)}, r_{(b+1,b)})$ for random $r_{(b,b+1)}, r_{(b+1,b)} \xleftarrow{R} \{0,1\}^\kappa$.

(b) $\mathcal{S}_0$ sends $(\mathsf{HIST}_{(0,1)}, \mathsf{HIST}_{(0,2)}, \mathsf{HIST}_{(2,1)}, r_{(0,1)}, r_{(0,2)})$ to $\mathcal{S}_1$. $\mathcal{S}_1$ sends $(\mathsf{HIST}_{(1,2)}, \mathsf{HIST}_{(1,0)}, \mathsf{HIST}_{(2,0)}, r_{(1,2)}, r_{(1,0)})$ to $\mathcal{S}_0$. $\mathcal{S}_2$ broadcasts $(r_{(2,0)}, r_{(2,1)})$.

(c) $\mathcal{S}_0$ and $\mathcal{S}_1$ verify the above hashes. If any of the hashes fail then the servers abort. Else, they perform the following:

$$\mathcal{S}_0 \text{ and } \mathcal{S}_1 \text{ compute } \mathsf{HIST}_0 \coloneqq \mathsf{HIST}_{(0,1)} + \mathsf{HIST}_{(1,0)}, \mathsf{HIST}_1 \coloneqq \mathsf{HIST}_{(1,2)} + \mathsf{HIST}_{(2,1)}, \text{ and } \mathsf{HIST}_2 \coloneqq \mathsf{HIST}_{(2,0)} + \mathsf{HIST}_{(0,2)}$$

(d) $\mathcal{S}_0$ and $\mathcal{S}_1$ abort if $\mathsf{HIST}_0 \neq \mathsf{HIST}_1$ or $\mathsf{HIST}_1 \neq \mathsf{HIST}_2$. Else, they output HIST where $\mathsf{HIST} = \mathsf{HIST}_0 = \mathsf{HIST}_1 = \mathsf{HIST}_2$.

Fig. 7: Private Histogram Protocol π_{HIST}.

consistency across the three sessions. To ensure this, the servers match the difference of the reconstructed output of $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ session, and the difference of the reconstructed output of $\mathcal{S}_2 - \mathcal{S}_0$ and $\mathcal{S}_1 - \mathcal{S}_2$ session, to verify that they are all 0. By transitivity, it is ensured that if and only if this check passes then the output of the VDPF evaluation would be the same across the three sessions, ensuring that $\alpha_1 = \alpha_2 = \alpha_3$.

A malicious server could collude with malicious clients. It can be observed that the honest clients' inputs are always hidden from the adversary due to input privacy of VDPF, since no server possesses more than one VDPF key. Next, A malicious server could attempt to incorporate an erroneous VDPF evaluation (from a malformed client input key) or inject additive errors into the output. We show how this is tackled in the protocol based on the server corruption:

- $\mathcal{S}_0$ *is corrupt.* In this case, the session between $\mathcal{S}_1 - \mathcal{S}_2$ is honest. $\mathcal{S}_0$ runs this session with $\mathcal{S}_1$ since it obtained $\mathsf{key}_{(2,1)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_0$ is supposed to send. This forces $\mathcal{S}_0$ to act honestly in the $\mathcal{S}_1 - \mathcal{S}_2$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_0$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_2 - \mathcal{S}_0$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_0$ could compute an incorrect hash $h := \mathrm{H}(\mathbf{Y}'_{(0,1)} - \mathbf{Y}'_{(0,2)} \parallel \mathbf{Y}'_{(0,2)} - \mathbf{Y}_{(2,1)})$, where $\mathbf{Y}'_{(0,1)}$ and $\mathbf{Y}'_{(0,2)}$ are incorrect. This would allow the $\mathcal{S}_0$ to introduce an additive error into the histogram (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ sessions) by incorporating the client's malformed input into the output histogram. However, this gets detected when the output histogram is reconstructed for all three sessions and compared. The output of $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ sessions will not match with the output of $\mathcal{S}_1 - \mathcal{S}_2$ session denoting that one of the servers behaved maliciously, hence leading to an abort.
- $\mathcal{S}_1$ *is corrupt.* This case is very similar to the above one where $\mathcal{S}_0$ was corrupt. In this case, the session between $\mathcal{S}_2 - \mathcal{S}_0$ is honest. $\mathcal{S}_1$ runs this session with $\mathcal{S}_0$ since it obtained $\mathsf{key}_{(2,0)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_1$ is supposed to send. This forces $\mathcal{S}_1$ to act honestly in the $\mathcal{S}_2 - \mathcal{S}_0$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_1$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_1 - \mathcal{S}_2$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_1$ simply ignore the hash $h := \mathrm{H}(\mathbf{Y}'_{(0,1)} - \mathbf{Y}_{(0,2)} \parallel \mathbf{Y}_{(0,2)} - \mathbf{Y}'_{(2,1)})$ (where $\mathbf{Y}'_{(0,1)}$ and $\mathbf{Y}'_{(2,1)}$ are incorrect) sent by $\mathcal{S}_0$. This would allow the $\mathcal{S}_1$ to introduce an additive error into the histogram (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_1 - \mathcal{S}_2$ sessions) by incorporating the client's malformed input into the output histogram. However, this gets detected when the output histogram is reconstructed for all three sessions and compared. The output of $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_1 - \mathcal{S}_2$ sessions will not match with the output of $\mathcal{S}_2 - \mathcal{S}_0$ session denoting that one of the servers behaved maliciously, hence leading to an abort.
- $\mathcal{S}_2$ *is corrupt.* In this case, the session between $\mathcal{S}_0 - \mathcal{S}_1$ is honest. If $\mathcal{S}_2$ behaves as a malicious attestator by sending incorrect hashes for the $\mathcal{S}_1 - \mathcal{S}_2$ or $\mathcal{S}_2 - \mathcal{S}_0$ sessions then the honest servers abort. Another way a malicious $\mathcal{S}_2$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in the three sessions. If the client provides malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1$ session then it gets detected due to verifiability of the VDPF, since both $\mathcal{S}_0$ and $\mathcal{S}_1$ are honest. It could provide malformed (allows double voting) VDPF keys $\mathsf{key}'_{(2,0)}$ and $\mathsf{key}'_{(2,1)}$ to $\mathcal{S}_1$ and $\mathcal{S}_0$ for the sessions involving $\mathcal{S}_2$. However, that again gets detected since the server $\mathcal{S}_0$ computes the hash h honestly and the $\mathcal{S}_1$ verifies it honestly.

5 Private Heavy Hitters

We provide the ideal functionality $\mathcal{F}_{\mathsf{HH}}$ for heavy-hitters computation between three servers and ℓ clients in Fig. 8. Adversary $\mathcal{A}$ maliciously corrupts any one of the servers and multiple clients.

Our detailed protocol π_{HH} that implements $\mathcal{F}_{\mathsf{HH}}$ appears in Figs. 10 and 11. High-level ideas of our protocol can be found in Sections 3.3 and 3.4. Our π_{HH} protocol privately computes all the $\mathcal{T}$-heavy-hitting strings (and their heavy-hitting prefixes) given the input data of ℓ clients, while protecting the privacy of the individual data points. π_{HH} runs on three servers $(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2)$ that utilize our verifiable incremental

Functionality $\mathcal{F}_{\mathsf{HH}}$

Parameters: Servers $\mathcal{S}_0, \mathcal{S}_1$ and $\mathcal{S}_2$, and ℓ clients $\mathcal{C}_i$ for $i \in [\ell]$. Servers $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$ agree upon:

- A bound ℓ on the number of client submissions.
- A bound $\mathcal{T}$ on the threshold for heavy hitters.

Inputs:

- *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$: No input.
- *Clients* $\mathcal{C}_i$: A point $\alpha_i \in \{0,1\}^n$ for $i \in [\ell]$.

Outputs: Initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$. Repeat for length of k bits, where $k \in [0, 1, \ldots n-1]$ and for each prefix $p \in \mathsf{HH}^k$:

- Update $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup (p \parallel 0)$ if
$$\textstyle\sum_{i=1}^{\ell} \left|(\alpha_{i, \leq k+1} = (p \parallel 0))\right| \geq \mathcal{T}.$$
- Update $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup (p \parallel 1)$ if
$$\textstyle\sum_{i=1}^{\ell} \left|(\alpha_{i, \leq k+1} = (p \parallel 1))\right| \geq \mathcal{T}.$$

$\mathcal{F}_{\mathsf{HH}}$ outputs the following:

- *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$: Set of $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n}$.
- *Clients* $\mathcal{C}_i$: No output for $i \in [\ell]$.

Corruption: Adversary $\mathcal{A}$ maliciously corrupts one server and multiple clients together. If $\mathcal{A}$ instructs the functionality to abort by sending $\perp$, the functionality returns $\mathsf{HH}^{\leq n}$ to the adversary and the functionality instructs the honest servers to output $\perp$.

Fig. 8: The ideal $\mathcal{F}_{\mathsf{HH}}$ functionality for $\mathcal{T}$-heavy hitters.

DPF (VIDPF) protocol to privately aggregate the clients' data points. Specifically, π_{HH} runs three VIDPF sessions, which guarantees security against a malicious server. Our protocol proceeds in three phases: a client computation phase, a server computation phase, and an output phase.

Client Computation. This phase is similar to the *client computation* of our histogram protocol presented in Section 4, except that the clients use our VIDPF primitive to generate the keys instead of the VDPF primitive. Each client generates VIDPF keys and sends them to the three servers as shown in Fig. 1.

Functionality $\mathcal{F}_{\mathsf{CMP}}$

Inputs: Party P_0 has input $(a_0, b_0, c_0, d_0, e_0, \mathcal{T}_0)$, Party P_1 has input $(a_1, b_1, c_1, d_1, e_1, \mathcal{T}_1)$, and Party P_2 has input $(a_2, b_2, c_2, d_2, e_2, \mathcal{T}_2)$.

Outputs: Compute the following:

$$a = a_0 + a_1 + a_2, \quad b = b_0 + b_1 + b_2, \quad c = c_0 + c_1 + c_2$$

$$d = d_0 + d_1 + d_2, \quad e = e_0 + e_1 + e_2$$

- Abort if $\mathcal{T}_0 \neq \mathcal{T}_1 \neq \mathcal{T}_2$.
- Set $\mathcal{T} = \mathcal{T}_0$.
- Output 1 if $a = b = c = d = e$ and $a \geq \mathcal{T}$. Else, output 0.

Corruption: Adversary $\mathcal{A}$ maliciously corrupts one server. If $\mathcal{A}$ instructs the functionality to abort by sending $\perp$, the functionality instructs the honest servers to abort.

Fig. 9: The ideal $\mathcal{F}_{\mathsf{CMP}}$ functionality for comparison.

Server Computation. Each server first initializes a set of sets for heavy-hitter computation as $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ, $\mathsf{HH}^1, \ldots, \mathsf{HH}^n$ are empty sets and HH^k corresponds to the kth level. The servers start accepting VIDPF keys from the clients. As in our histogram protocol, $\mathcal{S}_2$ acts as an attesting server for the sessions involving keys $\mathsf{key}_{(2,0)}$ and $\mathsf{key}_{(2,1)}$ by sending hashes (depicted in Fig. 2). Next, for $k \in [n]$ the servers perform the following:

(a) *Initialization.* For each k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$, the servers initialize to 0 a $\mathsf{count}^{p\|0}$ (resp. $\mathsf{count}^{p\|1}$) variable for each session to keep track of the frequency of prefix $p \| 0$ (resp. $p \| 1$). Later, each server aggregates for each of the three sessions their additive shares of each frequency in their local count variables. This is necessary for pruning the nodes later.

(b) *VIDPF Evaluation.* Next, the servers retrieve the states from memory for VIDPF evaluation in all three sessions corresponding to prefix $p \in \{0,1\}^k$ for each client. These states are used to incrementally evaluate the VIDPF on prefix strings $\gamma \in \{p \| 0, p \| 1\}$ for every client in all three sessions. For each client, the servers obtain new evaluation states (corresponding to prefix γ), VIDPF output for prefix string γ, and proof strings. The states are stored in the memory for future VIDPF evaluations on $\gamma \| 0$ and $\gamma \| 1$ in the $(k+1)$th level. More formally, the servers compute a secret shared vector $y^{\gamma}_{(b_1,b_2)}$ and a hash $\pi^{\gamma}_{(b_1,b_2)}$ that is used for consistency checking by relying on the verifiability property of the VIDPF. Next, the servers validate the client's input. If $k = 1$, then the servers reconstruct y^0 and y^1 for each client to verify that $y^0 + y^1 = 1$. If $k \neq 1$, then the servers reconstruct $y^p - (y^{p\|0} + y^{p\|1})$ and verify that it is 0. This ensures that the subtrees involving $p \| 0$ and $p \| 1$ are valid. The servers also need to ensure that the client has provided a consistent input across the three sessions. This is ensured by computing the difference of the reconstructed outputs across the sessions and verifying that they equal to 0 by matching their hash values $(\widehat{h^{p\|0}})$ and $(\widehat{h^{p\|1}})$ respectively, where

$$\widehat{h^{p\|0}} = \mathrm{H}_1(y^{p\|0}_{(0,1)} - y^{p\|0}_{(0,2)}, y^{p\|0}_{(0,2)} - y^{p\|0}_{(2,1)}),$$

$$\widehat{h^{p\|1}} = \mathrm{H}_1(y^{p\|1}_{(0,1)} - y^{p\|1}_{(0,2)}, y^{p\|1}_{(0,2)} - y^{p\|1}_{(2,1)}).$$

(c) *Batch-Verification.* The servers perform a batch verification of the hashes and the $y^p - (y^{p\|0} + y^{p\|1})$ values of all the clients succinctly by hashing them together and comparing the hashes. This is performed for all three sessions. If a client's VIDPF output is validated then they proceed to the aggregation phase, else their VIDPF output is ignored.

(d) *Aggregation.* Once a client's VIDPF output y^{γ} is validated for $\gamma \in \{p \| 0, p \| 1\}$, it is aggregated into $\mathsf{count}^{\gamma} = \mathsf{count}^{\gamma} + y^{\gamma}$. This is locally performed by each server (for all three sessions) using the secret shares of y^{γ} since it only involves addition. The servers perform this over every validated client output, and at the end of this phase, the servers possess a secret share of the frequency of $p \| 0$ and $p \| 1$ as $\mathsf{count}^{p\|0}$ and $\mathsf{count}^{p\|1}$.

(e) *Pruning.* The servers proceed to the pruning phase where they invoke $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 9) on the secret shares of count^{γ} (for $\gamma \in \{p \| 0, p \| 1\}$) for all three sessions and threshold $\mathcal{T}$. Based on the output of $\mathcal{F}_{\mathsf{CMP}}$ the following occurs:
   - $\mathcal{F}_{\mathsf{CMP}}$ returns 1 if $\mathsf{count}^{\gamma} \geq \mathcal{T}$ (i.e., γ is a heavy-hitter string). In this case, the prefix γ is added to the list of $k+1$-bit heavy-hitter set (i.e., $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \gamma$).
   - $\mathcal{F}_{\mathsf{CMP}}$ returns 0 if $\mathsf{count}^{\gamma} < \mathcal{T}$ (i.e., γ is a non heavy-hitter string). In this case, the prefix γ is ignored.
   - If $\mathcal{F}_{\mathsf{CMP}}$ returns $\perp$, which means that one of the servers behaved maliciously and it was detected, the servers abort.

   This computation is performed in parallel for all $(k+1)$-bit prefixes in consideration, and after the pruning phase, HH^{k+1} contains the list of $(k+1)$-bit heavy hitter strings. Next, the above computation is repeated for $(k+1)$-bit strings to compute $(k+2)$-bit heavy hitters, until we reach $k = n-1$. The comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [48].

Output Phase. At the end of the protocol, the servers output $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots, \mathsf{HH}^n\}$ as the set of $\mathcal{T}$-heavy hitter strings.

This completes the description of our protocol π_{HH} (Figs. 10 and 11).

The security of our protocol is captured in Theorem 2. Formal protocol details can be found below.

Theorem 2. *Assuming VIDPF is a verifiable incremental DPF and* H *is a random oracle then* π_{HH} *(Figs. 10 and 11) implements the* $\mathcal{F}_{\mathsf{HH}}$ *functionality in the random oracle model against malicious corruption of one server and* $\widetilde{\ell} \leq \ell$ *clients.*

Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}

- **Input:** Each client $\mathcal{C}_i$ has an input point $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$.
- **Output:** The servers $\mathcal{S}_b$ (for $b \in \{0,1,2\}$) output the set of $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} := \mathcal{F}_{\mathsf{HH}}(\ell, \mathcal{T}, \{\alpha_i\}_{i\in[\ell]})$.
- **Primitive:** VIDPF := ($\mathsf{Gen}, \mathsf{EvalPrefix}, \mathsf{EvalNext}$) is a verifiable incremental distributed point function.

1: **Client $\mathcal{C}$ Computation.** **(Repeated for ℓ clients, each of which has their own private input α)**

(a) Client $\mathcal{C}$ with input α prepares three pairs DPF keys with independent randomness $u, v, w \xleftarrow{R} \{0,1\}^\kappa$, as follows:

$$(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1, 1 \ldots, 1), \mathbb{G}), \quad (\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1, 1 \ldots, 1), \mathbb{G}),$$

$$(\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1, 1 \ldots, 1), \mathbb{G})$$

(b) The client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$.

2: **Server Computation.**

- The servers initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
- Repeat the following steps for length of k bits, where $k \in [0, \ldots, n-1]$:

(a) **Initialization.** For prefix $p \in \mathsf{HH}^k_b$, servers initialize the aggregation variables for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows:

$$\mathcal{S}_0 \text{ sets } \mathsf{count}^\gamma_{(0,1)} := \mathsf{count}^\gamma_{(0,2)} := \mathsf{count}^\gamma_{(2,1)} := 0, \quad \mathcal{S}_1 \text{ sets } \mathsf{count}^\gamma_{(1,2)} := \mathsf{count}^\gamma_{(1,0)} := \mathsf{count}^\gamma_{(2,0)} := 0$$

$$\mathcal{S}_2 \text{ sets } \mathsf{count}^\gamma_{(2,0)} := \mathsf{count}^\gamma_{(2,1)} := 0$$

(b) **VIDPFEvaluation.** For prefix $p \in \mathsf{HH}^{\leq k}_b$, Server $\mathcal{S}_b$ computes: **(Repeated for ℓ clients)**

i. Each server $\mathcal{S}_b$ retrieves the following states corresponding to the internal states of π_{VIDPF} computation for prefix p:

$$\mathcal{S}_0 \text{ retrieves } (\mathsf{st}^p_{(0,1)}, y^p_{(0,1)}, \pi^p_{(0,1)}), (\mathsf{st}^p_{(0,2)}, y^p_{(0,2)}, \pi^p_{(0,2)}) \text{ and } (\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$$

$$\mathcal{S}_1 \text{ retrieves } (\mathsf{st}^p_{(1,2)}, y^p_{(1,2)}, \pi^p_{(1,2)}), (\mathsf{st}^p_{(1,0)}, y^p_{(1,0)}, \pi^p_{(1,0)}) \text{ and } (\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$$

$$\mathcal{S}_2 \text{ retrieves } (\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)}) \text{ and } (\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$$

ii. Each server $\mathcal{S}_b$ evaluates the VIDPFon the prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows:

$$\mathcal{S}_0 \text{ computes } (\mathsf{st}^\gamma_{(0,1)}, y^\gamma_{(0,1)}, \pi^\gamma_{(0,1)}) := \mathsf{EvalPrefix}(0, \mathsf{key}_{(0,1)}, \gamma, \mathsf{st}^p_{(0,1)}, k, \pi^p_{(0,1)}) \text{ and stores it in memory.}$$

$$\mathcal{S}_0 \text{ computes } (\mathsf{st}^\gamma_{(0,2)}, y^\gamma_{(0,2)}, \pi^\gamma_{(0,2)}) := \mathsf{EvalPrefix}(1, \mathsf{key}_{(0,2)}, \gamma, \mathsf{st}^p_{(0,2)}, k, \pi^p_{(0,2)}) \text{ and stores it in memory.}$$

$$\mathcal{S}_1 \text{ computes } (\mathsf{st}^\gamma_{(1,2)}, y^\gamma_{(1,2)}, \pi^\gamma_{(1,2)}) := \mathsf{EvalPrefix}(0, \mathsf{key}_{(1,2)}, \gamma, \mathsf{st}^p_{(1,2)}, k, \pi^p_{(1,2)}) \text{ and stores it in memory.}$$

$$\mathcal{S}_1 \text{ computes } (\mathsf{st}^\gamma_{(1,0)}, y^\gamma_{(1,0)}, \pi^\gamma_{(1,0)}) := \mathsf{EvalPrefix}(1, \mathsf{key}_{(1,0)}, \gamma, \mathsf{st}^p_{(1,0)}, k, \pi^p_{(1,0)}) \text{ and stores it in memory.}$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_1 \text{ compute } (\mathsf{st}^\gamma_{(2,0)}, y^\gamma_{(2,0)}, \pi^\gamma_{(2,0)}) := \mathsf{EvalPrefix}(0, \mathsf{key}_{(2,0)}, \gamma, \mathsf{st}^p_{(2,0)}, k, \pi^p_{(2,0)}) \text{ and store them in memory.}$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_0 \text{ compute } (\mathsf{st}^\gamma_{(2,1)}, y^\gamma_{(2,1)}, \pi^\gamma_{(2,1)}) := \mathsf{EvalPrefix}(1, \mathsf{key}_{(2,1)}, \gamma, \mathsf{st}^p_{(2,1)}, k, \pi^p_{(2,1)}) \text{ and store them in memory.}$$

iii. If $k = 1$: Each server computes the proof that the VIDPFevaluation at the layer sums up to 1:

$$\mathcal{S}_0 \text{ computes } h^\emptyset_{(0,1)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(0,1)} - y^1_{(0,1)}) \text{ and } h^\emptyset_{(0,2)} := \mathrm{H}_1(\emptyset, y^0_{(0,2)} + y^1_{(0,2)})$$

$$\mathcal{S}_1 \text{ computes } h^\emptyset_{(1,2)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(1,2)} - y^1_{(1,2)}) \text{ and } h^\emptyset_{(1,0)} := \mathrm{H}_1(\emptyset, y^0_{(1,0)} + y^1_{(1,0)})$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_1 \text{ compute } h^\emptyset_{(2,0)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(2,0)} - y^1_{(2,0)}), \quad \mathcal{S}_2 \text{ and } \mathcal{S}_0 \text{ compute } h^\emptyset_{(2,1)} := \mathrm{H}_1(\emptyset, y^0_{(2,1)} - y^1_{(2,1)})$$

iv. If $k \neq 1$: Each server computes the proof that the VIDPFevaluation value along prefix p is same as the VIDPFevaluation value along the children, i.e., for $p \parallel 0$ and $p \parallel 1$.

$$\mathcal{S}_0 \text{ computes } h^p_{(0,1)} := \mathrm{H}_1(p, y^p_{(0,1)} - y^{p\parallel 0}_{(0,1)} - y^{p\parallel 1}_{(0,1)}) \text{ and } h^p_{(0,2)} := \mathrm{H}_1(p, -(y^p_{(0,2)} - y^{p\parallel 0}_{(0,2)} - y^{p\parallel 1}_{(0,2)}))$$

$$\mathcal{S}_1 \text{ computes } h^p_{(1,2)} := \mathrm{H}_1(p, y^p_{(1,2)} - y^{p\parallel 0}_{(1,2)} - y^{p\parallel 1}_{(1,2)}) \text{ and } h^p_{(1,0)} := \mathrm{H}_1(p, -(y^p_{(1,0)} - y^{p\parallel 0}_{(1,0)} - y^{p\parallel 1}_{(1,0)}))$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_1 \text{ compute } h^p_{(2,0)} := \mathrm{H}_1(p, y^p_{(2,0)} - y^{p\parallel 0}_{(2,0)} - y^{p\parallel 1}_{(2,0)}),$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_0 \text{ compute } h^p_{(2,1)} := \mathrm{H}_1(p, -(y^p_{(2,1)} - y^{p\parallel 0}_{(2,1)} - y^{p\parallel 1}_{(2,1)})).$$

$$\mathcal{S}_0 \text{ also sends hashes } (\widehat{h^{p\parallel 0}}, \widehat{h^{p\parallel 1}}) \text{ to } \mathcal{S}_1, \text{ where}$$

$$\widehat{h^{p\parallel 0}} = \mathrm{H}_1(y^{p\parallel 0}_{(0,1)} - y^{p\parallel 0}_{(0,2)}, y^{p\parallel 0}_{(0,2)} - y^{p\parallel 0}_{(2,1)}) \text{ and } \widehat{h^{p\parallel 1}} = \mathrm{H}_1(y^{p\parallel 1}_{(0,1)} - y^{p\parallel 1}_{(0,2)}, y^{p\parallel 1}_{(0,2)} - y^{p\parallel 1}_{(2,1)}).$$

Fig. 10: **Private Heavy Hitters Protocol π_{HH}** (continues in Fig. 11).

2: **Server Computation (Continued from Fig. 10)**

– (Cont.) Repeat the following steps for length of k bits, where $k \in [n]$:

(c) **Batch-Verification.** Verify VIDPFevaluations for all k-length prefixes p (and $p \parallel 0$ and $p \parallel 1$): **(Repeated for ℓ clients)**

$$\mathcal{S}_0 \text{ computes } R^k_{(0,1)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(0,1)}, \pi^{p\|0}_{(0,1)}, \pi^{p\|1}_{(0,1)})\Big) \text{ and } R^k_{(0,2)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(0,2)}, \pi^{p\|0}_{(0,2)}, \pi^{p\|1}_{(0,2)})\Big)$$

$$\mathcal{S}_1 \text{ computes } R^k_{(1,2)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(1,2)}, \pi^{p\|0}_{(1,2)}, \pi^{p\|1}_{(1,2)})\Big) \text{ and } R^k_{(1,0)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(1,0)}, \pi^{p\|0}_{(1,0)}, \pi^{p\|1}_{(1,0)})\Big)$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_1 \text{ compute } R^k_{(2,0)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(2,0)}, \pi^{p\|0}_{(2,0)}, \pi^{p\|1}_{(2,0)})\Big)$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_0 \text{ compute } R^k_{(2,1)} \coloneqq \mathrm{H}_2\Big(\big|\big|_{p \in \mathsf{HH}^k}\,(p, h^p_{(2,1)}, \pi^{p\|0}_{(2,1)}, \pi^{p\|1}_{(2,1)})\Big)$$

$\mathcal{S}_0$ sends $(R^k_{(0,1)}, R^k_{(0,2)}, R^k_{(2,1)})$ to $\mathcal{S}_1$. $\mathcal{S}_1$ sends $(R^k_{(1,2)}, R^k_{(1,0)}, R^k_{(2,0)})$ to $\mathcal{S}_0$. $\mathcal{S}_2$ sends $\widetilde{h^k_0} \coloneqq \mathrm{H}_3(R^k_{(2,0)})$ to $\mathcal{S}_0$ and $\widetilde{h^k_1} \coloneqq \mathrm{H}_3(R^k_{(2,1)})$ to $\mathcal{S}_1$.

Each server $\mathcal{S}_b$ sets $\mathsf{ver}_b \coloneqq 1$ for $b \in \{0,1\}$. $\mathcal{S}_b$ sets $\mathsf{ver}_b \coloneqq 0$ if $\widetilde{h^k_b}$ sent by $\mathcal{S}_2$ fail to match. Then, the servers locally perform the following computation:

$$\mathcal{S}_0 \text{ sets } \mathsf{ver}_0 \coloneqq 0 \text{ if } (R^k_{(0,1)} \neq R^k_{(1,0)}) \vee (R^k_{(0,2)} \neq R^k_{(2,0)}) \vee (R^k_{(2,1)} \neq R^k_{(1,2)})$$

$$\mathcal{S}_1 \text{ sets } \mathsf{ver}_1 \coloneqq 0 \text{ if } (R^k_{(1,2)} \neq R^k_{(2,1)}) \vee (R^k_{(1,0)} \neq R^k_{(0,1)}) \vee (R^k_{(2,0)} \neq R^k_{(0,2)}) \vee$$
$$(\widehat{h^{p\|0}} \neq \mathrm{H}_1(y^{p\|0}_{(2,0)} - y^{p\|0}_{(1,0)}, y^{p\|0}_{(1,2)} - y^{p\|0}_{(2,0)})) \vee (\widehat{h^{p\|1}} \neq \mathrm{H}_1(y^{p\|1}_{(2,0)} - y^{p\|1}_{(1,0)}, y^{p\|1}_{(1,2)} - y^{p\|1}_{(2,0)}))$$

$\mathcal{S}_0, \mathcal{S}_1$ broadcast the ver_0 and ver_1 bits. If any of the ver_0 or ver_1 is 0 then ignore this client's input.

(d) **Aggregation.** Aggregate the VIDPFoutputs for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: **(Repeated for ℓ clients)**

$$\mathcal{S}_0 \text{ sets } \mathsf{count}^\gamma_{(0,1)} \coloneqq \mathsf{count}^\gamma_{(0,1)} + y^\gamma_{(0,1)}, \mathsf{count}^\gamma_{(0,2)} \coloneqq \mathsf{count}^\gamma_{(0,2)} + y^\gamma_{(0,2)}, \text{ and } \mathsf{count}^\gamma_{(2,1)} \coloneqq \mathsf{count}^\gamma_{(2,1)} + y^\gamma_{(2,1)}$$

$$\mathcal{S}_1 \text{ sets } \mathsf{count}^\gamma_{(1,2)} \coloneqq \mathsf{count}^\gamma_{(1,2)} + y^\gamma_{(1,2)}, \mathsf{count}^\gamma_{(1,0)} \coloneqq \mathsf{count}^\gamma_{(1,0)} + y^\gamma_{(1,0)}, \text{ and } \mathsf{count}^\gamma_{(2,0)} \coloneqq \mathsf{count}^\gamma_{(2,0)} + y^\gamma_{(2,0)}$$

$$\mathcal{S}_2 \text{ sets } \mathsf{count}^\gamma_{(2,0)} \coloneqq \mathsf{count}^\gamma_{(2,0)} + y^\gamma_{(2,0)} \text{ and } \mathsf{count}^\gamma_{(2,1)} \coloneqq \mathsf{count}^\gamma_{(2,1)} + y^\gamma_{(2,1)}$$

The servers have aggregated the VIDPFevaluations (over all the ℓ clients) for all candidate $(k+1)$-bit strings.

(e) **Pruning.** Prune the non-heavy hitter strings. For every $(k+1)$-bit string γ, the servers perform the following:

- The servers invoke $\mathcal{F}_{\mathsf{CMP}}$ functionality (Fig. 9) with the additive shares of the node frequency. $\mathcal{F}_{\mathsf{CMP}}$ reconstructs the individual frequencies, and returns 1 if all the reconstructed frequencies match and the frequency is more than $\mathcal{T}$, else it returns 0.

$$\mathcal{S}_0 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{count}^\gamma_{(0,1)}, 0, \mathsf{count}^\gamma_{(0,2)}, \mathsf{count}^\gamma_{(2,1)}, \mathsf{count}^\gamma_{(0,2)}, \mathcal{T})$$

$$\mathcal{S}_1 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{count}^\gamma_{(1,0)}, \mathsf{count}^\gamma_{(1,2)}, 0, \mathsf{count}^\gamma_{(1,2)}, \mathsf{count}^\gamma_{(2,0)}, \mathcal{T})$$

$$\mathcal{S}_2 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(0, \mathsf{count}^\gamma_{(2,1)}, \mathsf{count}^\gamma_{(2,0)}, 0, 0, \mathcal{T})$$

  The servers abort if $\mathcal{F}_{\mathsf{CMP}}$ aborts. If $\mathcal{F}_{\mathsf{CMP}}$ outputs 1 then the servers include γ to the heavy-hitter set as $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \gamma$. Else, if the output is 0 then ignore γ since it is a non-heavy hitter string.

The servers have successfully computed the HH^{k+1} set. The servers repeat the *"Server Computation"* steps (starting from VIDPFevaluation for every client) for $k+1$ bit prefixes.

3: **Output Phase.** The servers output $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings.

Fig. 11: **Private Heavy Hitters Protocol π_{HH}** (continuing from Fig. 10).

Proof Sketch. The adversary is allowed to corrupt $\ell' \leq \ell$ clients and one of the servers. The rest two servers remain uncorrupted. We discuss the ways a malicious client can attempt to inject an error and we demonstrate our consistency checks for them:

- *Client VIDPF keys are malformed.* A malicious client can attempt to provide malformed VIDPF keys which are non-zero in more than one path in the binary tree (of 2^n leaves). This gets detected in the session involving the honest servers due to the verifiable property of the VIDPF at each level when the servers verify the proofs generated during the VIDPF evaluation. If the checks pass, then it is ensured that the VIDPF keys provided by the client are valid.
- *Client VIDPF input is malformed.* Next, a malicious client can try to double-vote on an input point, say $p \parallel 0 \in \{0,1\}^{k+1}$ by constructing the VIDPF on $(p \parallel 0, \widetilde{\beta^k})$. i.e., $f(p \parallel 0) = \widetilde{\beta^k}$, where $\widetilde{\beta^k} > 1$, instead of $(p \parallel 0, 1)$. This is detected by the honest servers since the honest servers perform a local subtree verification by reconstructing the value $y^p - (y^{p\parallel 0} - y^{p\parallel 1})$ and verifying that it equals 0 for all $k > 0$. For the base case, i.e., $k = 0$, the servers verify that $y^\epsilon = 1$. Combining all k checks ensures that $y^{p\parallel 0} = 1$ if and only if $y^p = 1$ and $y^{p\parallel 1} = 0$, else $y^{p\parallel 0} = 0$.
- *VIDPF input is inconsistent across sessions.* Finally, a malicious client can try to provide different VIDPF keys in different sessions, for example it constructs VIDPF keys for input $(\alpha_1, 1)$ for the $\mathcal{S}_0 - \mathcal{S}_1$ session and $(\alpha_2, 1)$ for the $\mathcal{S}_1 - \mathcal{S}_2$ session and $(\alpha_3, 1)$ for the $\mathcal{S}_2 - \mathcal{S}_0$ session, where $\alpha_1 \neq \alpha_2 \neq \alpha_3$ and $\alpha_1, \alpha_2, \alpha_3 \in \{0,1\}^k$. The above two checks would still pass since they ensure client input validation within each session but not client input consistency across the three sessions. To ensure this the servers match the difference of the reconstructed output of $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ session, and the difference of the reconstructed output of $\mathcal{S}_2 - \mathcal{S}_0$ and $\mathcal{S}_1 - \mathcal{S}_2$ session, to verify that they are all 0. By transitivity, it is ensured that if and only if this check passes then the output of the VIDPF evaluation would be the same across the three sessions, ensuring that $\alpha_1 = \alpha_2 = \alpha_3$. This is performed by computing the $\widehat{h^{p\parallel 0}}$ and $\widehat{h^{p\parallel 1}}$ hashes for every heavy-hitting prefix p computed by π_{HH}.

A malicious server could collude with malicious clients. It can be observed that the honest clients' inputs are always hidden from the adversary due to input privacy of VIDPF, since no server possesses more than one VIDPF key. Next, A malicious server could attempt to incorporate an erroneous VIDPF evaluation (from a malformed client input key) or inject additive errors into the output. We show how this is tackled in the protocol based on the server corruption:

- $\mathcal{S}_0$ *is corrupt.* In this case, the session between $\mathcal{S}_1 - \mathcal{S}_2$ is honest. $\mathcal{S}_0$ runs this session with $\mathcal{S}_1$ since it obtained $\mathsf{key}_{(2,1)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_0$ is supposed to send. This forces $\mathcal{S}_0$ to act honestly in the $\mathcal{S}_1 - \mathcal{S}_2$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_0$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_2 - \mathcal{S}_0$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_0$ could compute an incorrect hash $\widehat{h^{p\parallel 0}} := \mathrm{H}_1(y^{p\parallel 0}_{(0,1)}{}' - y^{p\parallel 0}_{(0,2)}{}', y^{p\parallel 0}_{(0,2)}{}' - y^{p\parallel 0}_{(2,1)})$ and $\widehat{h^{p\parallel 1}} := \mathrm{H}_1(y^{p\parallel 1}_{(0,1)}{}' - y^{p\parallel 1}_{(0,2)}{}', y^{p\parallel 1}_{(0,2)}{}' - y^{p\parallel 1}_{(2,1)})$ where $y^{p\parallel 0}_{(0,1)}{}', y^{p\parallel 0}_{(0,2)}{}', y^{p\parallel 1}_{(0,1)}{}', y^{p\parallel 1}_{(0,2)}{}'$ are incorrect. This would allow $\mathcal{S}_0$ to introduce an additive error into the frequency for $p \parallel 0$ and $p \parallel 1$ (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ sessions) by incorporating the client's malformed input. However, this gets detected when the output count is secretly reconstructed by the $\mathcal{F}_{\mathsf{CMP}}$ functionality for all three sessions and compared. The reconstructed count won't match and the ideal functionality would return a $\perp$ message detecting that one of the servers behaved maliciously, leading to an abort in the π_{HH}.
- $\mathcal{S}_1$ *is corrupt.* This case is very similar to the above one where $\mathcal{S}_0$ was corrupt. In this case, the session between $\mathcal{S}_2 - \mathcal{S}_0$ is honest. $\mathcal{S}_1$ runs this session with $\mathcal{S}_0$ since it obtained $\mathsf{key}_{(2,0)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_1$ is supposed to send. This forces $\mathcal{S}_1$ to act honestly in the $\mathcal{S}_2 - \mathcal{S}_0$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_1$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_1 - \mathcal{S}_2$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_1$ simply ignores the hash values $\widehat{h^{p\parallel 0}}$ and $\widehat{h^{p\parallel 1}}$ sent by $\mathcal{S}_0$. This would allow the $\mathcal{S}_1$ to introduce an additive

error into the frequency for $p \parallel 0$ and $p \parallel 1$ (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_1 - \mathcal{S}_2$ sessions) by incorporating the client's malformed input. However, this gets detected when the output count is secretly reconstructed by the $\mathcal{F}_{\mathsf{CMP}}$ functionality for all three sessions and compared. The reconstructed count won't match and the ideal functionality would return a $\perp$ message detecting that one of the servers behaved maliciously, leading to an abort in the π_{HH}.

– $\mathcal{S}_2$ *is corrupt.* In this case, the session between $\mathcal{S}_0 - \mathcal{S}_1$ is honest. If $\mathcal{S}_2$ behaves as a malicious attestator by sending incorrect hashes for the $\mathcal{S}_1 - \mathcal{S}_2$ or $\mathcal{S}_2 - \mathcal{S}_0$ sessions then the honest servers abort. Another way a malicious $\mathcal{S}_2$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in the three sessions. If the client provides malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1$ session then it gets detected due to verifiability of the VIDPF and the local subtree verification, since both $\mathcal{S}_0$ and $\mathcal{S}_1$ are honest. It could provide malformed (allows double voting) VIDPF keys $\mathsf{key}'_{(2,0)}$ and $\mathsf{key}'_{(2,1)}$ to $\mathcal{S}_1$ and $\mathcal{S}_0$ for the sessions involving $\mathcal{S}_2$. However, that again gets detected since the server $\mathcal{S}_0$ computes the hashes $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ honestly and the $\mathcal{S}_1$ verifies them honestly.

6 Optimizing Communication

We now present our novel optimization that enables our server-to-server communication to depend only on the number of malicious clients ℓ' and logarithmically on the total number of clients ℓ.

We perform a batched verification (for each server-to-server session) of all the clients' VDPF (resp. VIDPF) evaluations in the histogram protocol (resp. each level of the heavy-hitters protocol) between two servers using a Merkle tree with ℓ leaves. Our problem is equivalent to checking the equality of ℓ leaves between two servers, out of which ℓ' leaves can be malformed.

The servers hash their individual leaves and verify the equality of their Merkle tree roots. If the roots are equal then all the leaves are equal. Otherwise, if the roots are different, the servers verify the equality of the left children of the root node, and then the equality of the right children of the root node. If the left (resp. right) children are equal across the servers then the left (resp. right) subtree is equal. If the left (resp. right) children are different, then the servers apply the above algorithm to the left (resp. right) subtree. Proceeding this way in a recursive manner down the tree, the servers identify the malformed leaves where the two trees differ. This reduces our server-to-server communication for the consistency check to $\mathcal{O}(\kappa \log_2 \ell)$ bits if $\ell' = \mathcal{O}(1)$ clients behave maliciously. For ℓ' malicious clients, each server sends: $\mathcal{O}(\kappa) \times \min(\ell' \log \ell, (\ell - \ell') \log \frac{\ell}{\ell - \ell'})$ bits. We refer to page 10 of [43] for the detailed analysis of the expression.

7 Experimental Evaluations

We implement the PLASMA heavy-hitters protocol π_{HH} in Rust and use the `tarpc` framework by Google for asynchronous Remote Procedure Calls (RPC). PLASMA is fully parallelized: all sessions in each server run in parallel and we employ parallel iterators to process multiple client requests concurrently. We instantiate the PRG for VIDPF using the AES-NI hardware instructions for AES encryption with a seed length of $\kappa = 128$ bits. The group size for the intermediate levels of the VIDPF tree is 2^{62}, whereas for the leaves we use a finite field of $2^{2\kappa} = 2^{256}$ bits.

We report a series of experiments in which we vary both the number of clients (i.e., 10^3 to 10^6) and the size of the input strings (i.e., 32-bit to 256-bit words); 64-bit words are useful for our GPS application, while 256-bit words are useful for finding popular URLs. Additionally, we demonstrate the costs both from a client and a server perspective. For the clients, we measure the key generation time and the key size, while for the servers we show the communication size and the end-to-end runtime. Finally, we configure the threshold $\mathcal{T}$ to be more than 0.1% of the clients' strings.

7.1 Experimental Setup

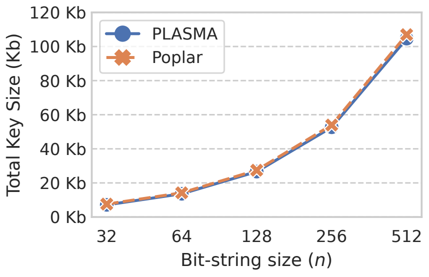
We evaluate PLASMA on AWS EC2 machines (c5.9xlarge) in the same region, each with 36 vCPUs at 3.60 GHz. PLASMA is compiled using Rust 1.61, and client-side experiments are carried out using a standard laptop with an Intel i7-8650U CPU (1.90 GHz).

7.2 Performance Evaluation

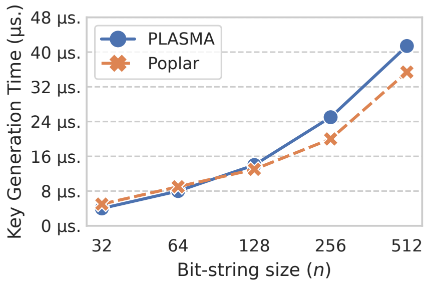
In our evaluation, our goal is to answer the following questions:

- How efficient is PLASMA for each client and server (with respect to computation and communication costs)?
- How does PLASMA compare with similar works (such as Poplar) that leverage DPFs?
- How does PLASMA compare with the related works that provide similar security guarantees, such as [5]?

7.2.1 Client costs First, we focus on the client-side costs, namely, the time for the key generation and the key sizes to be transmitted over the network to the servers. Recall that PLASMA requires three servers that run paired sessions ($\mathcal{S}_0$ and $\mathcal{S}_1$ run three sessions each and $\mathcal{S}_2$ acts as the consistency checking server) for a total of eight VDPF/VIDPF keys, whereas Poplar runs a single session on two servers for a total of two DPF keys. However, Poplar also includes an expensive malicious sketching operation that adds up to both the key sizes and the key generation time.
Key Size. In Fig. 12 (a) we compare the total size of all keys for all sessions that each client transmits to the servers for PLASMA and Poplar for increasing sizes of input strings. We observe that the client key size is almost identical between Poplar and PLASMA for $32-512$ bits although PLASMA transmits eight keys.



(a) Key size.



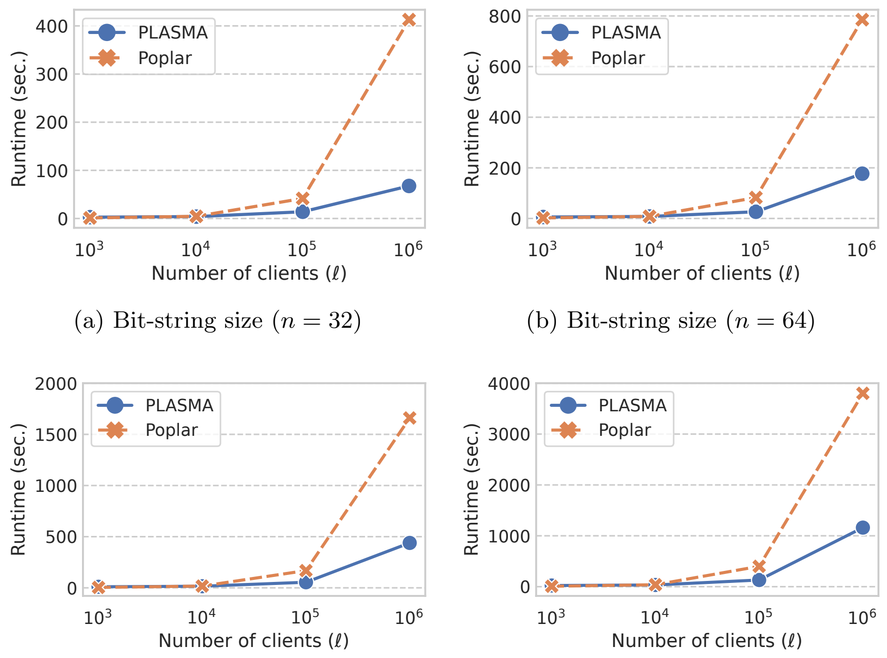
(b) Key generation time.

Fig. 12: Comparisons of client costs for PLASMA and Poplar. Key size in Kb and key generation time in microseconds (μs).

Key Generation Time. In Fig. 12 (b) we report the key generation time per client for PLASMA and Poplar for an increasing number of input bits n. Key generation is very fast even using modest hardware and PLASMA requires 45 microseconds for 512-bit inputs. For inputs smaller than 128 bits, the key generation time in the two frameworks takes approximately the same time, whereas Poplar is marginally faster than PLASMA for input size $n \geq 128$ bits.

7.2.2 Server costs Next, we study the server-side overhead of PLASMA and compare it with Poplar [16] and the sorting-based approach of [5]. Our experiments use an increasing number of clients from 10^3 to 10^6 with four different bit-string sizes, ranging from 32-bit words to 256-bit words, and statistical failure probability of 2^{-60}. We report our observations below.

Total Server Runtime: In Fig. 13 we present the total runtime for the PLASMA and Poplar servers to process all clients' requests from the moment they receive all client submissions. We observe that for less than 10^4 clients both PLASMA and Poplar perform very similarly. However, after increasing the number of clients, PLASMA outperforms Poplar by $3-6\times$: For $\ell = 10^5$ clients PLASMA is approximately $3\times$ faster



(a) Bit-string size ($n = 32$) (b) Bit-string size ($n = 64$)

(c) Bit-string size ($n = 128$) (d) Bit-string size ($n = 256$)

Fig. 13: Server runtime for an increasing number of clients and different word sizes.

than Poplar. Notably, this margin increases further as we go to 10^6 clients, where PLASMA is 6$\times$ faster than Poplar for 32-bit inputs and 3.2$\times$ faster for 256-bit inputs.

Although we observe some similarities on how the two frameworks scale due to their reliance on DPFs, PLASMA significantly outperforms Poplar as the number of clients increases, due to the expensive MPC sketching performed by Poplar. Conversely, PLASMA employs our efficient VIDPF construction. To fully understand the performance benefits of PLASMA, we performed additional experiments where we fixed the number of clients to 10^6 and completely removed the computation of the hashes from the intermediate levels of the DPF trees. By omitting the hashes for 32-bit inputs, PLASMA performs 2.5$\times$ faster, whereas for 256-bit inputs it performs 6.5$\times$ faster. Although hashes are generally computed very efficiently, in the case of multiple DPF levels (e.g., 256), each with many active paths, the hash computation becomes the bottleneck of PLASMA, highlighting the efficiency of the rest of our protocol. Meanwhile, the malicious secure sketching protocol of Poplar introduces an 8$\times$ slowdown on top of their semi-honest protocol.

Finally, for the sake of completeness, we run PLASMA in a similar setup as [5] and observe a 15$\times$ slowdown in server runtime.[4] However, these numbers are not comparable since [5] does not provide client input validation like PLASMA (which if implemented would add significant computational overheads). Moreover, [5] incurs 235$\times$ additional communication overhead, as discussed later.

Total Server-to-Server Communication: We compare the total communication costs incurred by all servers for an increasing number of clients and 256-bit strings in Fig. 14. We experimentally measured the cost of PLASMA and Poplar, whereas for the sorting-based approach of [5] we performed a detailed analysis in Appendix B to the best of our efforts. Both Poplar and the sorting-based approach incur significant communication overheads, whereas PLASMA achieves almost constant communication for an increasing number of clients. For reference, the total server-to-server communication cost for computing the heavy hitters over 10^6 clients is at least 45 gigabytes for [5] and 35 gigabytes for Poplar. Contrary, PLASMA

[4] The code of [5] is not publicly available; we use the runtimes presented in their paper.

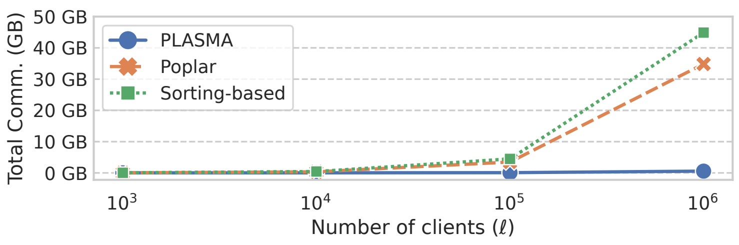


Fig. 14: Comparisons with Poplar [16] and the sorting-based approach of [5] in terms of total server-to-server communication (in GB), for bit-string size $n = 256$.

only requires a minimal communication of 200 megabytes for the same parameters. This yields a 182× improvement over Poplar and over 235× improvement over [5].

Applications. As discussed in Section 1.1, we evaluate PLASMA using two realistic applications as described below:

1. *Popular URLs.* In this standard benchmark for heavy hitters, each URL is represented as a 256-bit string. We report the total server runtime of PLASMA for detecting popular URLs in Fig. 13 (d) and the client communication costs in Figs. 12 (a) and (b) for $n = 256$. This benchmark is completed in under 20 minutes for 1 million clients with 200 MB of server-to-server communication, while Poplar incurs 3.3× additional runtime costs and 34.8 GBs of server-to-server communication.
2. *Popular GPS coordinates.* We employ *plus codes* [47] to efficiently encode the client GPS coordinates, instead of traditional latitude and longitude. This approach uses a grid system aligned on top of the world map, assigning specific codes to each area. The codes can be as short as two digits, indicating a general region, and by adding more digits, the position becomes more narrow. One advantage of using plus codes is that areas with similar codes are located in proximity to each other and a code that is a prefix of another encompasses the area of the latter. For instance, code `87` represents the North East US region, while code `87G8` represents a part of New York City. PLASMA uses plus codes to compute the most popular locations (submitted by more than $\mathcal{T} = 0.1\%$ of the clients) among a set of client-provided inputs using 64-bit strings in less than 3 minutes for 10^6 clients, as shown in Fig. 13 (b). Client cost is shown in Figs. 12 (a) and (b) for $n = 64$.

We further extend these applications by considering different heavy hitter thresholds based on some pre-agreed strings by the servers, which can be beneficial for traffic avoidance, since different roads may have different densities (e.g., highways are busier than smaller suburban roads). The servers can take that into consideration during evaluation and use higher $\mathcal{T}$s for highways with more vehicles, and lower thresholds for smaller roads. For completeness, we present our heavy hitters protocol for different thresholds in Appendix A.

8 Concluding Remarks

In this work, we present PLASMA: a framework to privately identify the most popular strings – or heavy hitters – among a set of client inputs without revealing the client data points. Previous works for private heavy hitters, such as Poplar, only considered security against malicious clients and were prone to additive attacks by a malicious server, compromising both the correctness and the security of the protocol. To address this challenge, PLASMA introduces a novel hash-based primitive, called verifiable incremental distributed point function, which allows the servers to validate client inputs and preemptively reject malformed ones. Additionally, we introduce a new batched consistency check that drastically reduces the communication cost for servers, by having the server-to-server communication depend only on the number of malicious clients, rather than the total number of clients. We have implemented PLASMA using Rust and report our evaluations for an increasing number of clients and a variety of input string sizes. In our approach, the server-to-server communication depends only on the number of malicious clients, as opposed to the total number of clients, yielding a 182× and 235× improvement over Poplar and other state-of-the-art sorting-based protocols respectively. PLASMA computes the 1000 most popular strings among a set of 1 million client-held 32-bit strings in 67 seconds and 256-bit strings in less than 20 minutes.

References

1. Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N Smart, T Tanguy, et al. Scale–mamba v1. 12: Documentation, 2021.
2. Erik Anderson, Melissa Chase, F. Betul Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Report 2021/1490, 2021. https://eprint.iacr.org/2021/1490.
3. Benny Applebaum, Eliran Kachlon, and Arpita Patra. Verifiable Relation Sharing and Multi-verifier Zero-Knowledge in Two Rounds: Trading NIZKs with Honest Majority - (Extended Abstract). In *Advances in Cryptology - CRYPTO 2022*, volume 13510 of *Lecture Notes in Computer Science*, pages 33–56, Santa Barbara, CA, USA, 2022. Springer.
4. Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 610–629, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.
5. Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 125–138, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.
6. Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. *Advances in Neural Information Processing Systems*, 30:1–32, 2017.
7. 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*, STOC '15, page 127–135, New York, NY, USA, 2015. Association for Computing Machinery.
8. Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 127–135, Portland, OR, USA, June 14–17, 2015. ACM Press.
9. K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.
10. Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 329–358, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.
11. James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. Distributed, private, sparse histograms in the two-server model. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 307–321, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.
12. Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 2025–2038, Virtual Event, USA, November 9–13, 2020. ACM Press.
13. Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems*, pages 59–74, Cham, 2014. Springer International Publishing.
14. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

15. Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2361–2377, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.
16. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
17. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.
18. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1292–1303, Vienna, Austria, October 24–28, 2016. ACM Press.
19. Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. https://eprint.iacr.org/2020/599.
20. 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.
21. Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2075–2092, London, UK, November 11–15, 2019. ACM Press.
22. Ran Canetti, Pratik Sarkar, and Xiao Wang. Blazing fast OT for three-round UC OT extension. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12111 of *Lecture Notes in Computer Science*, pages 299–327, Edinburgh, UK, May 4–7, 2020. Springer, Heidelberg, Germany.
23. Ran Canetti, Pratik Sarkar, and Xiao Wang. Efficient and round-optimal oblivious transfer and commitment with adaptive security. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 277–308, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.
24. Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. *IACR Cryptol. ePrint Arch.*, page 1212, 2020. (Accepted in Asiacrypt'22).
25. T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. Differentially private continual monitoring of heavy hitters from distributed streams. In Simone Fischer-Hübner and Matthew K. Wright, editors, *PETS 2012: 12th International Symposium on Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*, pages 140–159, Vigo, Spain, July 11–13, 2012. Springer, Heidelberg, Germany.
26. Shuchi Chawla, Cynthia Dwork, Frank McSherry, and Kunal Talwar. On Privacy-Preserving Histograms. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, page 120–127, Arlington, Virginia, USA, 2005. AUAI Press.
27. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.
28. Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 464–475, Berlin, Germany, 2003. VLDB Endowment.
29. Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 259–282, USA, 2017. USENIX Association.
30. Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 502–534, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.
31. Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A Private Time-Series Database from Function Secret Sharing. In *43rd IEEE Symposium on Security and Privacy, SP*, pages 2450–2468, San Francisco, CA, USA, 2022. IEEE.

32. Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 150–179, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.
33. Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 1054–1067, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.
34. Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *Proc. Priv. Enhancing Technol.*, 2016(3):41–61, 2016.
35. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.
36. Chaya Ganesh, Yashvanth Kondi, Arpita Patra, and Pratik Sarkar. Efficient adaptively secure zero-knowledge from garbled circuits. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 499–529, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany.
37. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.
38. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.
39. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 12: 15th International Conference on Information Security and Cryptology*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216, Seoul, Korea, November 28–30, 2013. Springer, Heidelberg, Germany.
40. Justin Hsu, Sanjeev Khanna, and Aaron Roth. Distributed Private Heavy Hitters. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'12, page 461–472, Berlin, Heidelberg, 2012. Springer-Verlag.
41. Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *EuroS&P*, pages 370–389, Genoa, Italy, 2020. IEEE.
42. Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Paper 2022/1561, 2022. https://eprint.iacr.org/2022/1561.
43. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
44. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1575–1590, Virtual Event, USA, November 9–13, 2020. ACM Press.
45. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.
46. Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. Private join and compute from PIR with default. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 605–634, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.
47. Google LLC. Open Location Code. https://github.com/google/open-location-code, 2019.
48. Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 249–270, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.
49. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang,

and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2111–2128, London, UK, November 11–15, 2019. ACM Press.

50. Dimitris Mouris, Charles Gouert, and Nektarios Georgios Tsoutsos. Privacy-Preserving IP Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2010–2023, 2022.
51. Dimitris Mouris, Charles Gouert, and Nektarios Georgios Tsoutsos. zk-Sherlock: Exposing Hardware Trojans in Zero-Knowledge. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 170–175, 2022.
52. Dimitris Mouris and Nektarios Georgios Tsoutsos. Pythia: Intellectual Property Verification in Zero-Knowledge. In *57th ACM/IEEE Design Automation Conference, DAC 2020, July 20-24, 2020*, pages 1–6, San Francisco, CA, USA, 2020. IEEE.
53. Dimitris Mouris and Nektarios Georgios Tsoutsos. Masquerade: Verifiable multi-party aggregation with secure multiplicative commitments. Cryptology ePrint Archive, Report 2021/1370, 2021. https://eprint.iacr.org/2021/1370.
54. Dimitris Mouris and Nektarios Georgios Tsoutsos. Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284, 2021.
55. Moni Naor, Benny Pinkas, and Eyal Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1369–1386, London, UK, November 11–15, 2019. ACM Press.
56. Arpita Patra, Pratik Sarkar, and Ajith Suresh. Fast actively secure OT extension for short secrets. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
57. 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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 192–203, Vienna, Austria, October 24–28, 2016. ACM Press.
58. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2986–3001, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.
59. Kang Yang and Xiao Wang. Non-interactive zero-knowledge proofs to multiple verifiers. Cryptology ePrint Archive, Report 2022/063, 2022. https://eprint.iacr.org/2022/063.
60. Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1607–1626, Virtual Event, USA, November 9–13, 2020. ACM Press.
61. Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated Heavy Hitters Discovery with Differential Privacy. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 3837–3847, Online, 26–28 Aug 2020. PMLR.

A Heavy Hitters with different Thresholds

Our protocol allows us to consider different heavy hitter thresholds $\mathcal{T}_i$ based on some pre-agreed strings $x_i \in \mathbf{X}$ by the servers. This can be beneficial for traffic avoidance since different roads may have different traffic densities. For example, highways are busier than smaller suburban roads. The servers can take that into consideration during evaluation, and use higher $\mathcal{T}$s for highways (since there are more vehicles), and lower thresholds for smaller roads.

We present our algorithm to compute heavy-hitters with different thresholds $\mathcal{T}_i$ for string $x_i \in \mathbf{X}$ from $\mathcal{T}$-prefix oracle query in Fig. 15. The prefix oracle query with different thresholds can be computed using a simple modification to protocol π_{HH}, where the pruning at the leaf layer is performed based on the threshold $\mathcal{T}_i$ for a given string $x_i \in \mathbf{X}$ instead of a fixed threshold $\mathcal{T}$.

```
Different Threshold Heavy Hitters from $\mathcal{T}$-prefix count queries

Parameters: Threshold $\mathcal{T}_i \in \mathbb{N}$, for string $x_i \in \mathbf{X}$ where $|\mathbf{X}| = m$, and string length $n \in \mathbb{N}$.
Inputs: The algorithm has no explicit input. It has access to $t$-prefix count query oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,t)$ for securely computing
$t$-prefix-count queries over prefix $p$ for strings $\alpha_1,\ldots,\alpha_\ell$.
Outputs: The set of heavy-hitter strings in $\alpha_1,\alpha_2,\ldots,\alpha_\ell$.
Algorithm:

– Initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where $\mathsf{HH}^0$ contains empty string $\epsilon$ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
– Set $\mathcal{T} = \min(\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_m)$.
– For each prefix $p \in \mathsf{HH}^k$ of length $k$-bits in set $\mathsf{HH}^k$ (where $k = 0,1,2,\ldots n-2$) :
    • If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \{p \parallel 0\}$.
    • If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \{p \parallel 1\}$.
– For each prefix $p \in \mathsf{HH}^{n-1}$, perform the following:
    • If $\exists x_i \in \mathbf{X}$ such that $(p \parallel 0) = x_i$ and $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}_i) = 1$, then set $\mathsf{HH}^n := \mathsf{HH}^n \cup (p \parallel 0)$.
    • If $\exists x_i \in \mathbf{X}$ such that $(p \parallel 1) = x_i$ and $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}_i) = 1$, then set $\mathsf{HH}^n := \mathsf{HH}^n \cup (p \parallel 1)$.
– Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.
```

Fig. 15: Algorithm for computing heavy hitters with different thresholds from $\mathcal{T}$-prefix count queries.

B Communication Cost of [5]

We now analyze the total server-to-server communication cost for the sorting-based protocol of [5]. We start from the optimized semi-honest communication cost from Appendix A.3 of [5], shown below:

$$mn(\frac{7}{3} + \frac{32}{9}||R||) + 3m||R|| + 2m||R'|| \text{ bits.}$$

We ignore the R' term since it is a payload. For malicious security, the protocol requires two times the semi-honest protocol, and additionally, the ring needs to be a field of size 2^κ size for $2^{-\kappa}$ failure probability. This leads us to the optimized malicious sorting protocol communication cost of:

$$2mn(\frac{7}{3} + \frac{32}{9}\kappa) + 3m\kappa.$$

The heavy hitters protocol requires the following for each item out of the total m items:

- Compute two secure comparisons over n bits. Assuming the state-of-the-art secure comparison protocol of Rabbit [48, Fig. 6], we get $\geq 4mn \log n$ from `LTBits` and `BitAdder` as well as mn to open the values.
- One secure multiplication over two secret shared n-bit variables: For m values it would be at least mn bits.
- Secure shuffling over and n-bit secret shared value, where the semi-honest shuffling takes $2m$ field element communication.

For malicious security, we consider the compiler of Chida et al. [27] and the communication cost is $2\times$ the semi-honest cost:

$$2(4mn \log n + mn + 2mn) = 8mn \cdot \log n + 6mn.$$

The per-server communication cost for their maliciously secure heavy-hitters protocol is at least:

$$2mn(\frac{7}{3} + \frac{32}{9} \cdot \kappa) + 3m\kappa + 8mn \log n + 6mn \text{ bits.}$$

Setting the security parameter κ to 60 bits, the number of items m to 10^6, and the number of bits of each item n to 256 bits we get that the communication cost should be at least:

$$2 \cdot 10^6 \cdot 256(\frac{7}{3} + \frac{32}{9}60) + 3 \cdot 10^6 \cdot 60 + (8 \cdot 10^6 \cdot 256 \cdot \log 256 + 6 \cdot 10^6 \cdot 256) = 14.96 \text{ giga bytes}$$

Therefore, the total server-server communication cost is at least $14.96 \cdot 3 \approx 45$ gigabytes for computing the heavy hitters over 256-bit keys between three servers for 10^6 clients.