PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries

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. Private heavy-hitters is a data-collection task where multiple clients possess private bit strings, and data-collection servers aim to identify the most popular strings without learning anything about the clients' inputs. In this work, we introduce PLASMA: a private analytics framework in the three-server setting that protects the privacy of honest clients and the correctness of the protocol against a coalition of malicious clients and a malicious server.
Our core primitives are a verifiable incremental distributed point function (VIDPF) and a batched consistency check, which are of independent interest. Our VIDPF introduces new methods to validate client inputs based on hashing. Meanwhile, our batched consistency check uses Merkle trees to validate multiple client sessions together in a batch. This drastically reduces server communication across multiple client sessions, resulting in significantly less communication compared to related works. Finally, we compare PLASMA with the recent works of Asharov et al. (CCS'22) and Poplar (S&P'21) and compare in terms of monetary cost for different input sizes.

Keywords: Function secret sharing, histograms, heavy hitters, privacy-enhancing technologies, 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 analysis, compute statistics, expose patterns in user behaviors, and apply them to improve their products [24, 29, 32, 14, 36]. Common analysis practices 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 responses.

Private Histograms. When computing histograms, it is crucial to maintain client privacy, such as preventing data collection servers from inferring additional information about the clients. Existing solutions for privacy-preserving histograms solve this problem efficiently [17, 6, 10], given a relatively small number of buckets. Nevertheless, histograms are resource-intensive on the server side when the goal is to find 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, 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 ones.

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 [28, 10], to discovering commonly used passwords [35], learning new words typed by users and identifying frequently used emojis [25], to name a few. Private heavy-hitters allow computing these results while also preserving client privacy. Existing protocols (such as [35, 10, 2, 8]) 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.

Table 1: Threat model comparisons, client input validation, and server-to-server communication.

Protocol	**Correctness & Privacy Against Malicious Corruption**			**Client Input Validation**	**Low Server-to-Server Communication**	**No. of Servers**
	Clients	**Server**	**Server & Clients**			
DPF [11, 12, 27]	●	◐†	○	○	○	2+
Poplar (IDPF) [10]	●	◐†	○	●	○	2
Bucketization (DP) [2]	●	◐†	○	●	○	2-3
MPC-based [8]	◐‡	◐†	○	○	○	3
Sorting-based [4, 30]	●	●	●	●	○	3
PLASMA (this work)	●	●	●	●	●	3

† These works only preserve privacy against a malicious server but not correctness.
‡ [8] is susceptible to data poisoning attacks by malicious clients or malicious servers. Privacy of honest clients is preserved.

Different Approaches. The literature considers the setting where two or more servers collect client inputs and run the private heavy-hitters protocol. A notable approach is based on differential privacy (DP), and the current state-of-the-art is [2] (we discuss DP-based solutions in 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 [8]) employ general-purpose secure computation frameworks (e.g., MP-SPDZ [31], SCALE-MAMBA [1], Sharemind [7]), so these methods fall short in terms of practicality.

Thus, recent works introduced custom MPC-based techniques for private heavy-hitters [4, 30]. The underlying protocols perform secure sorting of client inputs under MPC [4, 30] 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 large server-to-server communication, with linear dependency on the total number of clients, where the concrete communication cost is large.

Distributed point functions (DPFs) [11] offer an alternative approach for private histograms. Informally, DPFs allow a client to send succinct shares of a point function corresponding to their private inputs to two or more servers. The servers then use these shares to locally evaluate the function over the entire input space and add the resulting outputs to obtain additive shares of a histogram.

Poplar [10] builds upon the DPF approach by introducing the notion of incremental DPF (IDPF), which is detailed in Appendix A. Poplar provides an IDPF-based solution for the private heavy-hitter in the two-server setting, and 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. However, Poplar requires additional checks to perform input validation against malicious clients, which causes the server-to-server communication to scale linearly with the total number of clients. As a result, their concrete server-to-server communication is large.

Motivation. Since all aforementioned solutions incur server-to-server communication that scales linearly with the number of clients, where the concrete communication cost is large, they are prohibitive for most real-world applications that require millions of clients for data collection. Hence, it is desirable that the concrete server-to-server communication is low, even for a large number of clients. Likewise, neither Poplar nor the DP-based solutions [2] tolerate additive attacks from a malicious server, which results in incorrect outputs when one of the servers does not follow the protocol steps. More formally, they fail to provide both correctness and privacy against the collusion of a malicious server and malicious clients. In this regard, we ask the following motivating question:

Can we obtain a private heavy-hitters protocol with low concrete server-to-server communication that is secure against malicious clients and a malicious server?

1.1 Our Contributions

We answer the aforementioned question by proposing PLASMA, a framework for private and lightweight statistics that provides security against a malicious server and malicious clients. Our main contributions are summarized as follows:

Verifiable incremental DPF (VIDPF). First, we introduce a new primitive called verifiable incremental DPF (VIDPF), which builds upon incremental DPFs (IDPF) [10] and verifiable DPFs (VDPF) [20]. VIDPF allows us to verify that clients' inputs are valid by relying on hashing while preserving the client's input privacy. We also propose a novel way to verify that the IDPF keys are *"one-hot"* - i.e. they have a single non-zero evaluation path (containing the same value along the path) by solely relying on hashing. This is of independent interest since it can be used to improve earlier results in [10, 18]. Previous protocols (e.g., [10, 19]) solved the same problem using Zero-Knowledge [9] or expensive malicious sketching protocols involving information-theoretic MACs [10, 18].

Batched Consistency Check. Next, we introduce a novel batched consistency check that allows us to drastically reduce the server-to-server communication. At a high level, we validate the inputs of ℓ clients using a Merkle tree and identify the malformed ones using logarithmic (in the total number of clients denoted as ℓ) communication. This optimization reduces the dependency of our server-to-server communication on the total number of clients from $\mathcal{O}(\ell)$ to $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ number of hashes where there are ℓ' malicious clients, yielding a concrete improvement over the state-of-the-art (as reported in our experiments), even in the presence of malicious clients. Our communication cost remains low even when a constant fraction (e.g., 10%) of the clients are malicious.

PLASMA framework. We combine these new primitives to construct PLASMA, a protocol for private histograms and heavy hitters in the three-server setting that guarantees security against a malicious server and malicious clients while maintaining low 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 regard to runtime by a factor of $2.1\times$ over WAN for $\mathcal{T} = 1\%$ of the clients. In the same setting, our batched consistency check optimization enables us to drastically outperform both Poplar and the sorting-based protocol of [4] in terms of server-to-server communication by a factor of $35\times$ and $45\times$, respectively. For these conditions, we further analyzed the monetary cost of PLASMA, [4], and Poplar and report that PLASMA is $1.2 - 1.4\times$ and $2.5 - 3.5\times$ cheaper than these works, respectively.

Applications. We evaluate PLASMA for two applications of private heavy hitters: one that detects frequently visited URLs and another that identifies popular coordinates.

Popular URLs. A prominent application (discussed both in [4] and [10]) 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 6), we consider $n = 256$ bits, which is sufficient for standard domain names. PLASMA computes the heavy hitter URLs that caused more than 1% of client browsers to crash. We perform the task in 22 minutes for 10^6 clients while incurring less than 1 GB of server-to-server communication ($1.82 in total cost).

Popular GPS coordinates. We demonstrate a new application where PLASMA identifies popular geographic locations without sacrificing user privacy. This can be beneficial with traffic avoidance, restaurant recommendations, as well as advertising (e.g., businesses may 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 during rush hour. This is possible by encoding GPS coordinates as 64-bit strings using *plus codes* [33]. We compute the heavy hitter plus codes for a threshold $\mathcal{T} = 1\%$ in roughly 4 minutes across 10^6 clients while incurring very minimal server-to-server communication, costing $0.41 as total monetary cost.

Extensions. We also discuss how to extend PLASMA to obtain fairness against a malicious adversary that corrupts one server and an arbitrary number of clients. 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, where different coordinates (e.g. highways and suburban roads) have different congestion thresholds.

1.2 Related Work

We now discuss relevant works for private heavy hitters. They can be classified into four main groups: those based on DPFs, those based on differential privacy (DP), those based on MPC sorting, and finally those based on general-purpose MPC. A comparison of our protocol with related works can be found in Table 1.

1.2.1 DPF-based Distributed point functions [11] 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 [10], 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. In contrast, PLASMA provides security against both malicious clients and malicious server by adding one additional 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 Differential Privacy-based There is also a body of work based on local DP and randomized responses to compute heavy hitters [37, 5, 38]. These techniques only involve a single server collecting data from clients. Therefore, 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 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, and similarly to Poplar, it can only guarantee privacy *without correctness* in the presence of a malicious server. In contrast, PLASMA focuses on exact statistics and provides both correctness and privacy against both malicious clients and one malicious server.

1.2.3 Sorting-based Recent works in [4, 30] 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. Notably, PLASMA achieves a 45× improvement in server-to-server communication compared to [4] as shown in Fig. 12 for $\mathcal{T} = 1\%$. Moreover, our PLASMA protocol allows 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 MPC in the honest majority [26, 16] or dishonest majority setting [31] to compute 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 [8] provides a generic MPC-based protocol for computing differentially private heavy hitters. They use MPC frameworks like MP-SPDZ [31] and SCALE-MAMBA [1] to achieve semi-honest and malicious security, but their solution suffers from high communication and slow runtime.

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, as well as ℓ clients. The clients provide inputs to the servers and the servers do not have any private input. We assume that an adversary $\mathcal{A}$ maliciously corrupts one of the servers and $\ell' < \ell$ 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 PLASMA servers preemptively reject any malformed client input before incorporating it into the computation.

Servers. Similarly, a malicious server may try to deviate from the protocol steps and attempt to learn private user inputs; PLASMA always protects input *privacy* against one malicious server. Another possible attack for a malicious server would be to over-influence or corrupt the result of the protocol. The semi-honest model does not protect correctness against malicious behavior by a server, which is problematic in real-world applications, like advertisement measurements [14] between two companies, where one company may benefit from reporting inflated measurements by introducing undetectable errors. Malicious security ensures that such malicious behaviors are caught. Therefore, parties are forced to behave honestly, hence fostering a transparent environment for computation. Poplar has this limitation while PLASMA protects *correctness* against a malicious server. Hence, PLASMA is *robust* against a malicious server, since it protects both correctness and privacy.

2.2 Notation

We denote the computational and statistical security parameters by κ and μ, respectively. 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 $\mathbf{X} \coloneqq \{x_1, x_2, \dots, x_m\}$ where the ith 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 $x_{i,\leq L} \coloneqq (x_{i,1}, x_{i,2}, \dots x_{i,L})$ for $L \leq n$. Let $\mathcal{S}_b$ denote the bth server, for $b \in \{0,1,2\}$; we consider $b+1 \coloneqq (b+1) \mod 3$ and $b+2 \coloneqq (b+2) \mod 3$. We assume ℓ clients, each denoted as $\mathcal{C}_i$ for $i \in [\ell]$. For an n-bit string a we represent its bit decomposition as $a_1, \dots, a_n \in \{0,1\}$. Each client $\mathcal{C}_i$ has an n-bit input string $\alpha_i \in \mathbf{X}$, for $i \in [\ell]$. We use $\alpha_{i,1}, \dots \alpha_{i,n} \in \{0,1\}$ to denote the bit representation of the client's input α_i.

2.3 Distributed Point Functions (DPF)

Function secret sharing (FSS) [11] enables splitting the output of a function f into additive shares, where each share of the function is represented by a separate key. Each key allows the owner to efficiently generate an additive share of the output $f(x)$ on a given input x. DPFs are a special case of FSS where f is a point function $f_{\alpha,\beta}(x) \coloneqq \beta$ if $x = \alpha$, or 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 evaluates 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). We discuss DPFs and other stronger notions, such as incremental DPFs (IDPF) [10] and verifiable DPFs (VDPF) [20], in Appendix A.

3 Technical Overview

We recall the histogram and heavy-hitters protocol by Poplar [10] in Section 3.1. Then, we briefly describe our histogram protocol in Section 3.2 as a stepping stone to our heavy-hitters protocol, which we describe in Sections 3.3 and 3.4.

3.1 Histogram Protocol of Poplar

Poplar first considers the problem of computing private subset histograms. Each client holds an n-bit string α and the servers $\mathcal{S}_0$, $\mathcal{S}_1$ have a small set $\mathbf{X} \coloneqq \{x_1, x_2, \dots, x_m\}$ of m n-bit strings. Each client secret shares their input α using a DPF as $(\mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathrm{DPF}.\mathsf{Gen}(1^\kappa, \alpha, 1, \mathbb{G})$. 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 $y_b \coloneqq \sum_{x_i \in \mathbf{X}} \mathrm{DPF}.\mathsf{Eval}(b, \mathsf{key}_b, x_i)$. 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, Poplar introduces a malicious sketching protocol to ensure that the client inputs are well-formed. 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's server-to-server communication scales linearly with $\mathcal{O}(\ell)$ due to the malicious sketching protocol.

3.2 Our Histogram Protocol

We address Poplar's limitations by (1) introducing one additional server, (2) building upon the primitive of verifiable DPF [20] (Appendix A), and (3) introducing novel consistency checks in the three-party setting. We claim the following benefits over Poplar:

(a) Robustness against a collusion of a malicious server and malicious clients,

(b) Lightweight consistency checks for malicious behavior (using only symmetric operations and field additions),

(c) Server-to-server communication depends logarithmically on the total number of clients.

In fact, our work provides the first maliciously secure protocol whose server-to-server communication is logarithmic in the total number ℓ of clients. Our servers communicate $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ hashes for the consistency checks, where ℓ' is the number of corrupt clients. Similar to Poplar, we ensure client input validation against malicious clients (i.e., honest servers preemptively detect inconsistent client input and discard it). Here, we present the ideas of our histogram protocol, which are crucial for our heavy-hitters protocol in Section 3.4.2.

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).

However, adding a third server significantly complicates things as we need to ensure consistency between the three sessions. For instance, we need to check that a malicious client submitted the same input α to all three sessions without revealing it. 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 one of the servers behaves maliciously (by launching an additive attack) and the honest servers abort, which provides robustness against the malicious server.

Motivation for three servers. The three-server model is widely considered both in the industry and academia as it ensures practical deployments with malicious security. Notable examples include the Interoperable Private Attribution (IPA) proposal by Meta and Mozilla [14], JP Morgan's PrimeMatch [36], NTT's heavy-hitters protocol [4], among others. The servers can be hosted by companies and non-profit organizations (e.g., as mentioned in Section 5 of Google-Apple's Covid Exposure system [3]). Table 1 compares our work with state-of-the-art results.

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.

In our optimization, instead of running two sessions in each server, we run all three sessions between $\mathcal{S}_0$ and $\mathcal{S}_1$ and use $\mathcal{S}_2$ as the attestation server. By doing that, we significantly reduce the latency due to the synchronization overhead of the three servers. To enable that, our 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 is illustrated in Fig. 1.

Our optimization allows $\mathcal{S}_0$ to replicate the computation of $\mathcal{S}_2$ 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 just sending hashes to $\mathcal{S}_1$ for the same messages that $\mathcal{S}_0$ should send. These hashes prevent $\mathcal{S}_0$ from acting maliciously. Similar protocol steps are run by $\mathcal{S}_2$ to attest the $\mathcal{S}_2 - \mathcal{S}_0$

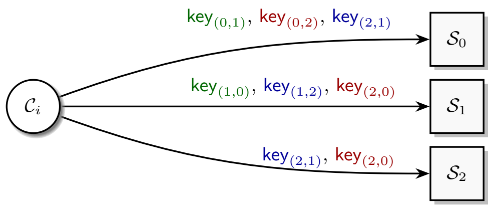

$\mathsf{key}_{(0,1)}$, $\mathsf{key}_{(0,2)}$, $\mathsf{key}_{(2,1)}$
$\mathcal{S}_0$
$\mathcal{C}_i$
$\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(1,2)}$, $\mathsf{key}_{(2,0)}$
$\mathcal{S}_1$
$\mathsf{key}_{(2,1)}$, $\mathsf{key}_{(2,0)}$
$\mathcal{S}_2$

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

session and prevent $\mathcal{S}_1$ (who is replicating $\mathcal{S}_2$) from acting maliciously. We summarize this attestation process in Fig. 2. Overall, this optimization allows us to batch-verify all three sessions as a single session between $\mathcal{S}_0$ and $\mathcal{S}_1$ using hashes.

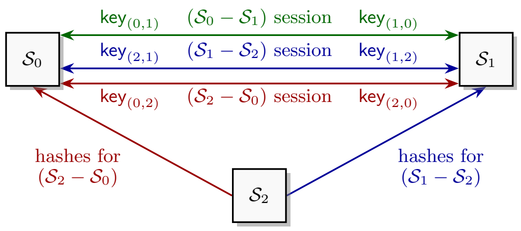

$\mathsf{key}_{(0,1)}$ $(\mathcal{S}_0 - \mathcal{S}_1)$ session $\mathsf{key}_{(1,0)}$
$\mathsf{key}_{(2,1)}$ $(\mathcal{S}_1 - \mathcal{S}_2)$ session $\mathsf{key}_{(1,2)}$
$\mathsf{key}_{(0,2)}$ $(\mathcal{S}_2 - \mathcal{S}_0)$ session $\mathsf{key}_{(2,0)}$
$\mathcal{S}_0$
$\mathcal{S}_1$
hashes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

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

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 this class of attacks, we propose a novel consistency check that only relies on inexpensive symmetric operations, like hashing.

We first ensure that the DPF output is non-zero only at a single point. The work of [20] introduces the primitive of verifiable DPF (VDPF), which we summarize in Appendix A. This is a stronger notion of DPF, where the servers obtain a correctness proof π upon evaluating a pair of DPF keys on a given input point. The two servers obtain the same proof π if the client generates the DPF keys honestly (i.e., the DPF output is non-zero only 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 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 well-formed (i.e., is not 1), then the client's input is discarded. If the output is 1 (i.e., the client behaved honestly), then the DPF output shares are aggregated by the server in the histogram share.

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 in each session respectively. The verifiability of the VDPF fails to detect this attack since each individual VDPF in each session is valid.

To address the challenge, we propose 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 the VDPF evaluation by $\mathcal{S}_0$ on keys $\mathsf{key}_{(0,1)}$, $\mathsf{key}_{(0,2)}$, and $\mathsf{key}_{(2,1)}$ corresponding to sessions $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, 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 the VDPF evaluation by $\mathcal{S}_1$ on keys $\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(2,0)}$, and $\mathsf{key}_{(1,2)}$ corresponding to sessions $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, 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 $h \coloneqq \mathrm{H}(\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} \parallel \mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)})$ to $\mathcal{S}_1$, who verifies that $h = \mathrm{H}(\mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)} \parallel \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)})$. 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. This leads to $\mathcal{O}(\kappa\ell)$ server-server communication for ℓ clients, but it is optimized to logarithmic communication by applying batched client verification, described in Section 5.

3.3 Heavy-Hitters from $\mathcal{T}$-Prefix Count

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 DPFs (summarized in Appendix A). 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 mitigate this leakage, we introduce the notion of $\mathcal{T}$-threshold prefix-count queries that return 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 Query Oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,\mathcal{T})$). *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.*

$\mathcal{T}$-Heavy hitters. The $\mathcal{T}$-heavy hitters algorithm (for threshold $\mathcal{T}$) is provided with oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,\mathcal{T})$ for computing $\mathcal{T}$-prefix count for prefix p over the client input strings $\alpha_1,\ldots,\alpha_\ell$. The initial prefix is 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 pruning, the algorithm outputs the set of strings that are $\mathcal{T}$-heavy hitters. Our formal algorithm is presented in Fig. 3.

$\mathcal{T}$-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:

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

Cost Analysis. 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 $\mathcal{T}$-Prefix Count Queries Oracle from VIDPF

We realize the $\mathcal{T}$-Prefix Count Query Oracle $\Omega(\cdot,\mathcal{T})$ from Def. 1 by relying on a new verifiable incremental DPF (VIDPF) primitive and using an ideal functionality $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 7) for secure comparison.

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 the party possessing key_b to process 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:

– $\mathsf{Gen}(1^\kappa, 1^n, \alpha, (\beta^1, \beta^2, \ldots, \beta^n), \mathbb{G}) \to (\mathsf{key}_0, \mathsf{key}_1)$: Given security parameter κ, input size n, input string $\alpha \in \{0,1\}^n$, and values $\beta^1, \ldots, \beta^n$, the key generation algorithm outputs two VIDPF keys key_0 and key_1.
– $\mathsf{EvalPref}(b, \mathsf{key}_b, x, \mathsf{st}_b^{k-1}, \pi_b^{k-1}) \to (\mathsf{st}_b^k, y_b, \pi_b^k)$: Given a VIDPF key key_b and an input string $x \in \{0,1\}^k$ of length $k \le n$ 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 $k \le n$, the following holds for all $k \le n$:

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

where $(\mathsf{st}_0^k, y_0, \pi_0^k) := \mathsf{EvalPref}(0, \mathsf{key}_0, x, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$ and $(\mathsf{st}_1^k, y_1, \pi_1^k) := \mathsf{EvalPref}(1, \mathsf{key}_1, x, \mathsf{st}_1^{k-1}, \pi_1^{k-1})$. 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 anything about the input α or the outputs $\beta^1, \ldots, \beta^n$ of the client.
– *Verifiability.* This 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 $(\mathsf{key}_0, \mathsf{key}_1)$ outputs $(\beta^1, \beta^2, \ldots, \beta^k)$. More formally, for any $k \in [n]$ 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:

$$(\mathsf{st}_0^k, y_0, \pi_0^k) \coloneqq \mathsf{EvalPref}(0, \mathsf{key}_0, z, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$$
$$(\mathsf{st}_1^k, y_1, \pi_1^k) \coloneqq \mathsf{EvalPref}(1, \mathsf{key}_1, z, \mathsf{st}_1^{k-1}, \pi_1^{k-1})$$
$$y_0 + y_1 = \begin{cases} \beta^k, \text{if } z = \widetilde{x}, \\ 0, \text{if } z = \{0,1\}^k \setminus \{\widetilde{x}\}, \end{cases}$$

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 $\mathsf{EvalPref}$ algorithm on $k-1$ bits of z. The evaluators initialize $\mathsf{st}_0^0 \coloneqq \mathsf{st}_1^0 \coloneqq 0$ and $\pi_0^0 \coloneqq \pi_1^0 \coloneqq 0$.

We provide a construction of VIDPF in Figs. 14 and 15 (Appendix B) based on length doubling PRG in the random oracle model. 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 keys, one for each pair of servers, with independent randomness but the same input point α and output values $(1, \ldots, 1)$. The client sends the keys for the sessions to the respective servers (Fig. 1) as in our histogram protocol.

Basic Protocol. As depicted in Fig. 2, $\mathcal{S}_1$ replicates $\mathcal{S}_2$ in the $\mathcal{S}_2 - \mathcal{S}_0$ session and $\mathcal{S}_2$ behaves as an attestator for $\mathcal{S}_1$ by sending hashes of the messages that $\mathcal{S}_1$ should send. The hash prevents server $\mathcal{S}_1$ from acting maliciously corresponding to the $\mathcal{S}_2 - \mathcal{S}_0$ session. Similar protocol steps are run by $\mathcal{S}_2$ for the session $\mathcal{S}_1 - \mathcal{S}_2$, 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 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. Checks 1-3 ensure that the VIDPF keys are *"one-hot"*, i.e., they have a single non-zero evaluation path (containing 1 in this case, along the path), and check 4 ensures that the client input is consistent across the sessions:

- *Check 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.
- *Check 2:* For the root level (i.e., $k = 0$), the servers evaluate the VIDPF keys on the empty string ϵ and verify it is 1.
- *Check 3:* Finally, at the k^{th} level, the servers need to verify that $y^{p\|0}$ is either 0 or 1, without reconstructing the output. We perform this check by observing that the output of the parent p should be the sum of the outputs of $p \,\|\, 0$ and $p \,\|\, 1$. The servers evaluate the VIDPF keys on the parent string p and sibling (of $p \,\|\, 0$) string $p \,\|\, 1$ 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. The first check ensures that at most one of $y^{p\|0}$ or $y^{p\|1}$ is non-zero. Combining the two checks, we can conclude that either $(y^{p\|0} = 0, y^{p\|1} = 1)$ or $(y^{p\|0} = 1, y^{p\|1} = 0)$, since at most one child can equal 1 when the parent holds a value of 1. Iterating this for all k levels ensures that $y^{p\|0} = 1$ iff $y^p = 1$ and $y^{p\|1} = 0$, else $y^{p\|0} = 0$. The servers also verify (using check 1) the corresponding proofs π generated during the VIDPF evaluation along the path, to ensure there is at most one non-zero path in the entire binary tree.
- *Check 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. For more details, we defer to Section 4.

Output Phase. Once the client's VIDPF output $y^{p\|0}$ is verified, the secret shares of $y^{p\|0}$ are aggregated into counter $\mathsf{cnt}^{y\|0}$. 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. 7) with the secret shares of cnt and threshold $\mathcal{T}$. $\mathcal{F}_{\mathsf{CMP}}$ reconstructs cnt and it outputs 1 if $\mathsf{cnt} \geq \mathcal{T}$, 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 $y \,\|\, 0$. The comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [34].

Robustness Against a Malicious Server. The third server ensures that if the client behaves 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, cnt is reconstructed across the three sessions by $\mathcal{F}_{\mathsf{CMP}}$. If cnt 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 causes the honest servers to abort, providing robustness against the malicious server. We observe that our protocol satisfies *fairness* (which is a stronger security notion than selective abort) if $\mathcal{F}_{\mathsf{CMP}}$ is implemented using a fair protocol. We discuss this in Section 7.

Batched Client Verification. In our final protocol, we verify multiple client inputs at each level in one batch. We batch all the clients' VIDPF evaluations using a Merkle tree that has ℓ leaves for ℓ clients. First, the servers check the equality of ℓ leaves by asserting that the Merkle roots are the same. If the roots match then the leaves are the same, while 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 the dependency of our server-to-server communication to $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$, for ℓ' malicious clients, instead of $\mathcal{O}(\ell)$. Formal details of this verification can be found in Section 5.

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* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$ do not have any input. *Clients* $\mathcal{C}_i$: A point $\alpha_i \in \{0,1\}^n$ for $i \in [\ell]$. $\alpha_{i,j}$ represents the jth bit of α_i.

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

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

$\mathcal{F}_{\mathsf{HH}}$ 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. $\mathcal{A}$ can perform the following:

- If $\mathcal{A}$ instructs the functionality to discard the jth client's input then $\mathcal{F}_{\mathsf{HH}}$ discards α_j from the output computation.
- If $\mathcal{A}$ instructs the functionality to abort at level $k+1$ by sending $(\perp, k+1)$, then the functionality returns $\mathsf{HH}^{\leq k}$ to $\mathcal{A}$ and the honest servers; additionally, the functionality instructs the honest servers to abort by sending $\perp$.

Fig. 4: The ideal $\mathcal{F}_{\mathsf{HH}}$ functionality for $\mathcal{T}$-heavy hitters.

4 Private Heavy Hitters

We provide the ideal functionality $\mathcal{F}_{\mathsf{HH}}$ for heavy-hitters between three servers and ℓ clients in Fig. 4. Adversary $\mathcal{A}$ maliciously corrupts any one of the servers and multiple clients. If $\mathcal{A}$ has maliciously corrupted a server, it can instruct $\mathcal{F}_{\mathsf{HH}}$ to discard an honest client's input as part of its adversarial behavior. It can also instruct the functionality to abort at a particular level $k+1$. In this case, $\mathcal{A}$ and the honest servers receive the set of all (that have not been discarded by $\mathcal{A}$) k-bit heavy-hitting prefixes as output, and the functionality instructs the honest servers to abort.

Our detailed protocol π_{HH} that implements $\mathcal{F}_{\mathsf{HH}}$ appears in Figs. 5 and 6, while 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 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. During the client computation phase, each client $\mathcal{C}$ prepares three pairs of VIDPF keys for their private data point $\alpha \in \mathbf{X}$, and output value $(\beta^1, \ldots, \beta^n) := (1, \ldots, 1)$ along the path to α, using independent randomness for each key generation. Employing three pairs of keys essentially allows us to run three separate VIDPF 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 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 $\mathsf{cnt}^{p\|0}$ (resp. $\mathsf{cnt}^{p\|1}$) variable for each session to count the frequency of prefix $p \,\|\, 0$ (resp. $p \,\|\, 1$). Each server aggregates

– **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 := $(\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

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:

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

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

(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 $p \in \mathsf{HH}_b^k$, servers initialize the aggregation variables for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows:

$$\mathcal{S}_0 \text{ sets } \mathsf{cnt}^\gamma_{(0,1)} := \mathsf{cnt}^\gamma_{(0,2)} := \mathsf{cnt}^\gamma_{(2,1)} := 0, \quad \mathcal{S}_1 \text{ sets } \mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0, \quad \mathcal{S}_2 \text{ sets } \mathsf{cnt}^\gamma_{(2,0)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$$

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

i. If $(p = \emptyset)$: then $\mathcal{S}_0$ sets $\mathsf{st}^\emptyset_{(0,1)} := \pi^\emptyset_{(0,1)} := \mathsf{st}^\emptyset_{(0,2)} := \pi^\emptyset_{(0,2)} := \mathsf{st}^\emptyset_{(2,1)} := \pi^\emptyset_{(2,1)} := \emptyset$, Server $\mathcal{S}_1$ sets $\mathsf{st}^\emptyset_{(1,2)} := \pi^\emptyset_{(1,2)} := \mathsf{st}^\emptyset_{(1,0)} := \pi^\emptyset_{(1,0)} := \mathsf{st}^\emptyset_{(2,0)} := \pi^\emptyset_{(2,0)} := \emptyset$. Server $\mathcal{S}_2$ sets $\mathsf{st}^\emptyset_{(2,0)} := \pi^\emptyset_{(2,0)} := \mathsf{st}^\emptyset_{(2,1)} := \pi^\emptyset_{(2,1)} := \emptyset$.

If $(p \neq \emptyset)$: then each server $\mathcal{S}_b$ retrieves the following states from memory corresponding to the internal states of π_{VIDPF} computation for prefix p: Server $\mathcal{S}_0$ retrieves $(\mathsf{st}^p_{(0,1)}, y^p_{(0,1)}, \pi^p_{(0,1)})$, $(\mathsf{st}^p_{(0,2)}, y^p_{(0,2)}, \pi^p_{(0,2)})$ and $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$. Server $\mathcal{S}_1$ retrieves $(\mathsf{st}^p_{(1,2)}, y^p_{(1,2)}, \pi^p_{(1,2)})$, $(\mathsf{st}^p_{(1,0)}, y^p_{(1,0)}, \pi^p_{(1,0)})$ and $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$. Server $\mathcal{S}_2$ retrieves $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$ and $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$.

ii. Each server $\mathcal{S}_b$ evaluates the VIDPF on the prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows and stores them in memory:

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

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

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

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

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

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

iii. If $k = 1$: Servers compute the proof that the VIDPF evaluation at the root layer sums up to 1:

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

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

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

iv. If $k \neq 1$: Servers compute proof that (VIDPF output on prefix p) = (VIDPF output on prefix $p \parallel 0$) + (VIDPF output on prefix $p \parallel 1$):

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

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

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

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

Fig. 5: **Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}** (continues in Fig. 6).

2: **Server Computation (Continued from Fig. 5)**

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

(b) **(Cont.) VIDPF Evaluation.**

v. $\mathcal{S}_0$ and $\mathcal{S}_1$ ensure that the client input is consistent across the three sessions by computing the following hashes.

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

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

vi. *Client State Accumulation:* The servers accumulate their local state for each client session as follows:

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

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

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

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

(c) **Batch-Verification.** The servers batch-verify the client inputs for all three sessions and across the three sessions by invoking π_{check} (Fig. 8):

i. $\mathcal{S}_0$ sets $u_i := \{(R^k_{(0,1)}, R^k_{(0,2)}, R^k_{(2,1)}, \widehat{h^{p\|0}}, \widehat{h^{p\|1}})$ values for client $i \in [\ell]\}$. $\mathcal{S}_1$ sets $v_i := \{(R^k_{(1,0)}, R^k_{(2,0)}, R^k_{(1,2)}, \overline{h^{p\|0}}, \overline{h^{p\|1}})$ values for client $i \in [\ell]\}$. $\mathcal{S}_0$ sets $\mathbf{u} := \{u_i\}_{i\in[\ell]}$ and $\mathcal{S}_1$ sets $\mathbf{v} := \{v_i\}_{i\in[\ell]}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ batch-verify all the client inputs by computing the bit ver and list L (comprising of invalid client inputs) by running π_{check} with inputs $\mathbf{u}$ and $\mathbf{v}$ respectively: $(\mathsf{ver}, \mathsf{L}) := \pi_{\mathsf{check}}(\mathbf{u}, \mathbf{v})$:

$$\mathsf{ver} := 0 \text{ if } \exists \text{ client whose } (R^k_{(0,1)} \neq R^k_{(1,0)}) \vee (R^k_{(0,2)} \neq R^k_{(2,0)}) \vee (R^k_{(2,1)} \neq R^k_{(1,2)}) \vee (\widehat{h^{p\|0}} \neq \overline{h^{p\|0}}) \vee (\widehat{h^{p\|1}} \neq \overline{h^{p\|1}}),$$

and $\mathsf{L} := \{$list of invalid clients' since they failed to pass the above check$\}$. If $\mathsf{ver} = 1$, then all the clients' inputs are valid.

ii. $\mathcal{S}_2$ possesses $R^k_{(2,0)}, R^k_{(2,1)}$ values for each client. $\mathcal{S}_2$ verifies that $\mathcal{S}_2$'s version of $R^k_{(2,1)}$ matches with $\mathcal{S}_0$'s version of $R^k_{(2,1)}$. $\mathcal{S}_2$ also attests that $\mathcal{S}_2$'s version of $R^k_{(2,0)}$ matches with $\mathcal{S}_0$'s version of $R^k_{(0,2)}$ by computing $(\mathsf{ver}', \mathsf{L}')$ as follows:

$$(\mathsf{ver}', \mathsf{L}') := \pi_{\mathsf{check}}(\{R^k_{(2,1)}, R^k_{(2,0)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_2, \{R^k_{(2,1)}, R^k_{(0,2)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_0).$$

iii. $\mathcal{S}_2$ verifies that $\mathcal{S}_2$'s version of $R^k_{(2,0)}$ matches with $\mathcal{S}_1$'s version of $R^k_{(2,0)}$. $\mathcal{S}_2$ also attests that $\mathcal{S}_2$'s version of $R^k_{(2,1)}$ matches with $\mathcal{S}_1$'s version of $R^k_{(1,2)}$ by computing $(\mathsf{ver}'', \mathsf{L}'')$ as follows:

$$(\mathsf{ver}'', \mathsf{L}'') := \pi_{\mathsf{check}}(\{R^k_{(2,0)}, R^k_{(2,1)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_2, \{R^k_{(2,0)}, R^k_{(1,2)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_0).$$

After batch verification, the servers identify the list of bad clients as $\mathsf{L} := \mathsf{L} \cup \mathsf{L}' \cup \mathsf{L}''$. The servers ignore the inputs of all clients in L.

(d) **Aggregation.** Aggregate the VIDPF outputs for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: **(Repeated for all validated clients in** $[\ell] \setminus \mathsf{L}$**)**

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

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

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

The servers have aggregated the VIDPF evaluations (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. 7) with the additive shares of the node frequency.

$$\mathcal{S}_0 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{cnt}^\gamma_{(0,1)}, 0, \mathsf{cnt}^\gamma_{(0,2)}, \mathsf{cnt}^\gamma_{(2,1)}, \mathsf{cnt}^\gamma_{(0,2)}, \mathcal{T}), \quad \mathcal{S}_1 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{cnt}^\gamma_{(1,0)}, \mathsf{cnt}^\gamma_{(1,2)}, 0, \mathsf{cnt}^\gamma_{(1,2)}, \mathsf{cnt}^\gamma_{(2,0)}, \mathcal{T}),$$

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

The servers abort if $\mathcal{F}_{\mathsf{CMP}}$ aborts. If $\mathcal{F}_{\mathsf{CMP}}$ outputs 1 set $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \gamma$. Else, the servers ignore γ since it is non-heavy hitter.

Servers have successfully computed the HH^{k+1} set. Servers repeat *"Server Computation"* steps (starting from Step 2b) on $k+1$ bit prefixes.

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

Fig. 6: **Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}** (continuing from Fig. 5).

for each of the three sessions their additive shares of each frequency in their local cnt variables and uses them for pruning.

(b) *VIDPF Evaluation.* Next, the servers retrieve from memory the states 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 $\gamma \in \{p \parallel 0, p \parallel 1\}$ 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 memory for future VIDPF evaluations on $\gamma \parallel 0$ and $\gamma \parallel 1$ 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 $y^0 + y^1$ for each client to verify that $y^0 + y^1 = 1$. If $k \neq 1$, then the servers reconstruct $y^p - (y^{p\parallel 0} + y^{p\parallel 1})$ and verify that it is 0. This ensures that the subtrees involving $p \parallel 0$ and $p \parallel 1$ 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 0 by matching their hash values with the other servers' hash in Step 2b(v) of Fig. 5.

(c) *Batch-Verification.* The servers need to check: (1) that the hashes they possess for a client are equal, and (2) that $y^p = (y^{p\parallel 0} + y^{p\parallel 1})$. Both these checks are reduced to checking the equality of a string (corresponding to each client) held by servers. Let $\mathbf{u}$ (resp. $\mathbf{v}$) be the list of ℓ (one for each client) strings held by the first (resp. second) server. Then, the servers perform a batch verification of $\mathbf{u}$ and $\mathbf{v}$ strings by invoking the subprotocol $\pi_{\mathsf{check}}(\mathbf{u}, \mathbf{v})$ in Fig. 8. If the two lists $\mathbf{u}$ and $\mathbf{v}$ are equal then π_{check} returns $\mathsf{ver} = 1$, else it returns $\mathsf{ver} = 0$ and a list L containing the indices of elements where the lists differ. This is performed for all three sessions. $\mathcal{S}_2$ also attests the sessions that it is involved in. This is performed using batch-verification, yielding output lists L' and L''. Finally, the servers identify the list of bad clients as $\mathsf{L} = \mathsf{L} \cup \mathsf{L}' \cup \mathsf{L}''$ and their VIDPF output is ignored. The servers consider the rest of the clients as "validated" and they are moved to the aggregation phase.

(d) *Aggregation.* Once a client's VIDPF output y^{γ} is validated for $\gamma \in \{p \parallel 0, p \parallel 1\}$, it is aggregated into $\mathsf{cnt}^{\gamma} := \mathsf{cnt}^{\gamma} + y^{\gamma}$. 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 $p \parallel 0$ and $p \parallel 1$ as $\mathsf{cnt}^{p\parallel 0}$ and $\mathsf{cnt}^{p\parallel 1}$.

(e) *Pruning.* The servers proceed to pruning and invoke $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 7) on the secret shares of cnt^{γ} (for $\gamma \in \{p \parallel 0, p \parallel 1\}$) for all sessions and threshold $\mathcal{T}$. Based on the output of $\mathcal{F}_{\mathsf{CMP}}$ the following occurs:
   - $\mathcal{F}_{\mathsf{CMP}}$ returns 1 if $\mathsf{cnt}^{\gamma} \geq \mathcal{T}$ (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 $\mathsf{cnt}^{\gamma} < \mathcal{T}$ (i.e., γ is a non heavy-hitter string). In this case, the prefix γ is ignored.
   - If $\mathcal{F}_{\mathsf{CMP}}$ returns $\perp$, then one of the servers behaved maliciously and the honest servers abort. This occurs if the malicious server has provided an incorrect threshold as input (condition 1 in $\mathcal{F}_{\mathsf{CMP}}$) or it provided incorrect client output shares as input (condition 4 in $\mathcal{F}_{\mathsf{CMP}}$).

   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 $k = n-1$. As already mentioned, $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [34].

Output Phase. At the end, 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 π_{HH} (Figs. 5, 6). Security of our protocol is captured in Theorem 1 and proven in Appendix C.

Theorem 1. *Assuming VIDPF is a verifiable incremental DPF and* $\mathrm{H}_1, \mathrm{H}_2$ *are random oracles,* $\mathcal{F}_{\mathsf{CMP}}$ *is a secure comparison functionality (Fig. 7), and* H *(in* π_{check}*) is collision-resistant, then* π_{HH} *(Figs. 5 and 6) implements* $\mathcal{F}_{\mathsf{HH}}$ *in the (random oracle,* $\mathcal{F}_{\mathsf{CMP}}$*)-model against malicious corruption of one server and* $\ell' \leq \ell$ *clients.*

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 $a \coloneqq a_0 + a_1, b \coloneqq b_1 + b_2, c \coloneqq c_0 + c_2$, $d \coloneqq d_0 + d_1, e \coloneqq e_1 + e_2$, and proceed as follows:

1. If $\mathcal{T}_0 \neq \mathcal{T}_1 \neq \mathcal{T}_2$, then $\mathcal{F}_{\mathsf{CMP}}$ aborts. Else, set $\mathcal{T} \coloneqq \mathcal{T}_0$.
2. If $a = b = c = d = e$ and $a < \mathcal{T}$ output 0.
3. If $a = b = c = d = e$ and $a \geq \mathcal{T}$ output 1.
4. Else, $\mathcal{F}_{\mathsf{CMP}}$ aborts (i.e. a, b, c, d or e strings are not equal).

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

π_{check}

Inputs: Party P_0 has ℓ input strings $\mathbf{u} = \{u_i\}_{i \in [\ell]}$. Party P_1 has ℓ input strings $\mathbf{v} = \{v_i\}_{i \in [\ell]}$.

Outputs: π_{check} outputs $(\mathsf{ver}, \mathsf{L})$ as follows:

- If $\mathbf{u} = \mathbf{v}$: $\mathsf{ver} \coloneqq 1$ and $\mathsf{L} \coloneqq \emptyset$,
- If $\mathbf{u} \neq \mathbf{v}$: $\mathsf{ver} \coloneqq 0$ and $\mathsf{L} \coloneqq \{i\}_{u_i \neq v_i \text{ for } i \in [\ell]}$.

$\mathsf{ver} = 1$ (resp. $\mathsf{ver} = 0$) denotes that the Merkle roots of $\mathbf{u}$ and $\mathbf{v}$ are equal (resp. unequal). List L is a list of indices where $\mathbf{u}$ and $\mathbf{v}$ differ.

Parameters: $\mathrm{H} : \{0,1\}^{\kappa} \to \{0,1\}^{\kappa}$ is a collision-resistant hash. $\mathsf{K} = \lceil \log_2 \ell \rceil$ denotes number of levels in the Merkle tree for ℓ leaves.

Algorithm:

Root Computation: Party P_0 (resp. P_1) locally computes the Merkle R_0 (resp. R_1) on $\mathbf{u}$ (resp. $\mathbf{v}$). For $b \in \{0,1\}$, party P_b performs:

- If $b = 0$: set $\mathsf{N}_0^{\mathsf{K}} \coloneqq \{\mathsf{N}_{0,i}^{\mathsf{K}}\}_{i \in [\ell]} \coloneqq \{\mathrm{H}(\mathsf{K}, i, u_i)\}_{i \in \ell}$ as the list of leaf nodes in the Merkle tree containing $\mathbf{u}$.
- If $b = 1$: set $\mathsf{N}_1^{\mathsf{K}} \coloneqq \{\mathsf{N}_{1,i}^{\mathsf{K}}\}_{i \in [\ell]} \coloneqq \{\mathrm{H}(\mathsf{K}, i, v_i)\}_{i \in \ell}$ as the list of leaf nodes in the Merkle tree containing $\mathbf{v}$.
- Initialize $\ell' \coloneqq \ell$ as the number of nodes in level K.
- For level $k \in \{\mathsf{K}-1, \mathsf{K}-2, \ldots, 1\}$:
  - Set $\ell' \coloneqq \lceil \frac{\ell'}{2} \rceil$ as the number of nodes in level k.
  - For $i \in [\ell']$: Compute list of nodes at level k by hashing the nodes at level $k+1$ as $\mathsf{N}_b^k \coloneqq \mathsf{N}_b^k \cup \mathrm{H}(k, \mathsf{N}_{b,2i}^{k+1}, \mathsf{N}_{b,2i+1}^{k+1})$.
- Set Merkle $\mathsf{R}_b \coloneqq \mathsf{N}_b^1$.

Root Verification: Parties P_0 and P_1 exchange R_0 and R_1. If $\mathsf{R}_0 = \mathsf{R}_1$ then set $\mathsf{ver} \coloneqq 1$, $\mathsf{L} \coloneqq \emptyset$, and parties output $(\mathsf{ver}, \mathsf{L})$. Else, set $\mathsf{ver} \coloneqq 0$ and continue the computation.

Unequal Leaf Identification: For $b \in \{0,1\}$, party P_b sets $\overline{\mathsf{N}}_b^1 \coloneqq \mathsf{R}_b$ as the unequal node at level 1.

- For level $k \in \{2, \ldots, \mathsf{K}\}$: For each unequal node $n \in \overline{\mathsf{N}}_b^{k-1}$ at level $k-1$, parties identify unequal nodes at level k:
  - Party P_b fetches left and right child of n as $\mathsf{child}_b^{\mathsf{L}}$ and $\mathsf{child}_b^{\mathsf{R}}$ respectively, for $b \in \{0,1\}$.
  - Parties exchange $\mathsf{child}_0^{\mathsf{L}}$, $\mathsf{child}_1^{\mathsf{L}}$, $\mathsf{child}_0^{\mathsf{R}}$ and $\mathsf{child}_1^{\mathsf{R}}$, and performs the following for $b \in \{0,1\}$:

$$\overline{\mathsf{N}}_b^k \coloneqq \overline{\mathsf{N}}_b^k \cup \mathsf{child}_b^{\mathsf{L}} \text{ if } \mathsf{child}_0^{\mathsf{L}} \neq \mathsf{child}_1^{\mathsf{L}}$$

$$\overline{\mathsf{N}}_b^k \coloneqq \overline{\mathsf{N}}_b^k \cup \mathsf{child}_b^{\mathsf{R}} \text{ if } \mathsf{child}_0^{\mathsf{R}} \neq \mathsf{child}_1^{\mathsf{R}}$$

P_b possesses $\overline{\mathsf{N}}_b^{\mathsf{K}}$ as list of unequal leaf nodes. P_b sets L as the list of indices of $\overline{\mathsf{N}}_b^{\mathsf{K}}$ w.r.t. initial leaf nodes $\mathsf{N}_b^{\mathsf{K}}$ as $\mathsf{L} \coloneqq \mathsf{L} \cup \{i : \overline{\mathsf{N}}_{b,i}^{\mathsf{K}} = \mathsf{N}_{b,i}^{\mathsf{K}}\}$.
Party P_b outputs $(\mathsf{ver}, \mathsf{L})$.

Fig. 8: Equality verification of ℓ strings between two parties and identification of unequal strings.

5 Batched Consistency Check

We now present our batched consistency check π_{check} that enables two parties, P_0 and P_1, to verify the equality of lists $\mathbf{u}$ and $\mathbf{v}$ containing ℓ strings using Merkle trees. If the two lists are equal then π_{check} returns $\mathsf{ver} = 1$, else it returns $\mathsf{ver} = 0$ and a list L containing the indices of elements where the lists differ. Correctness follows from the collision resistance property of the hash function H.

As summarized in Fig. 8, π_{check} requires $\mathsf{K} + 1$ rounds of communication, where $\mathsf{K} = \lceil \log_2 \ell \rceil$. The total communicated hashes are roughly $4\ell'(\log_2 \frac{\ell}{\ell'} + 2)$, where $\mathbf{u}$ and $\mathbf{v}$ differ on ℓ' elements. It can be further optimized to $2\ell'(\log_2 \frac{\ell}{\ell'} + 2)$, where only one of the parties sends its hashes instead of both. We provide a detailed analysis of the protocol in Appendix D.

6 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 apply the same parallelization for benchmarking Poplar.) We instantiate the PRG for VIDPF using the AES-NI hardware instructions for AES encryption with a seed of $\kappa = 128$ bits. The group size for intermediate levels is 2^{62}, whereas for the leaves we use a finite field of $2\kappa = 256$ bits. Notably, PLASMA can be implemented with rings instead of fields since our checks rely on the security of the VIDPF. On the other hand, the security of Poplar relies on the group size and needs 62 bits for the statistical failure probability to be 2^{-60}.

Experiment Details. Our experiments vary the number of clients between $\ell = 10^3$ and $\ell = 10^6$ with two different bit-string sizes, $n = 64$ and $n = 256$ bits. We configured the threshold $\mathcal{T}$ to be 1% of the clients' strings, and we report the client and server costs, while empirically comparing with Poplar. Then, we compute the total monetary costs (due to runtime and communication) incurred by PLASMA servers, and we compare it with [4] (since the code of [4] is not open-source) based on the monetary cost.

Experimental Setup. We performed both LAN and WAN[3] experiments on AWS EC2 machines (c5.9xlarge) 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).

Performance Evaluation. In our experiments, our goal is to answer the following questions:

- How efficient is PLASMA for each client and server?
- 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 [4]?

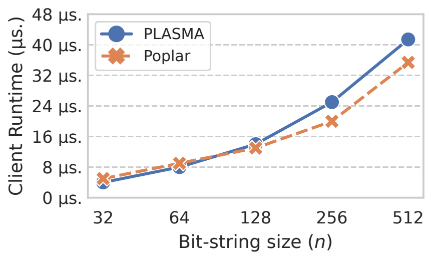
Client costs. The PLASMA client generates three pairs of DPF keys. Meanwhile, the Poplar client generates two pairs of DPF keys but also computes a malicious sketching operation. As a result, both PLASMA and Poplar clients are extremely fast, running in the order of $20 - 24$ microseconds on 256-bit inputs. A detailed comparison of client runtime can be found in Fig. 9 (a).

In terms of client communication, PLASMA transmits eight DPF keys, whereas Poplar transmits four DPF keys plus the correlated randomness for the sketching operation. We observed that the clients in both protocols incur the same communication overhead, roughly around 55 KB for 256 bits. Detailed comparisons can be found in Fig. 9 (b).

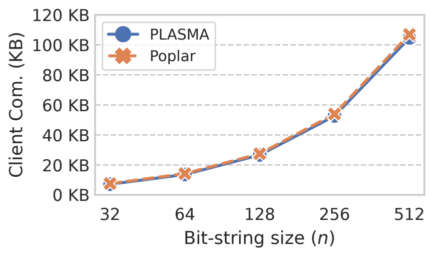
Server costs. In this experiment, we run PLASMA with randomly distributed malicious clients and compare it with Poplar. The number of malicious clients ℓ' for PLASMA is 0%, 1%, and 10% of the total clients ℓ.

Server Runtime over LAN. PLASMA outperforms Poplar in terms of server runtime by $1.1\times$ (64 bits) and $2.1\times$ (256 bits) for $\ell = 10^6$ clients and $\mathcal{T} = 1\%$ of the clients. Notably, this improvement in PLASMA is largely attributed to our efficient VIDPF-based client input validation and remains mostly unaffected even in the presence of malicious clients, as presented in Fig. 10. Meanwhile, Poplar servers validate clients' inputs using an expensive malicious secure sketching protocol.

[3] We used one server in Oregon, one in Ohio, and one in N. Virginia. For Poplar, we used one in Oregon and the other one in N. Virginia.

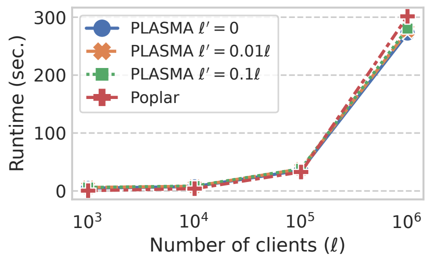


(a) Client Runtime

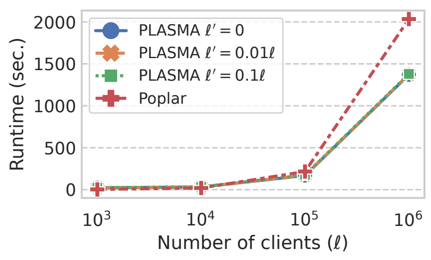


(b) Client Communication

Fig. 9: Comparisons of client costs for PLASMA and Poplar (KB is Kilobytes and μs is microseconds).



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



(b) Bit-string size ($n = 256$)

Fig. 10: Server runtime (over LAN) for an increasing number of clients.

Server Runtime over WAN. We benchmarked PLASMA and Poplar over WAN for $n = 64$ bits and we report our findings in Fig. 11. While the total latency is increased for both frameworks, we observe that the server WAN runtime for PLASMA increased by roughly 10% compared to server LAN runtime, whereas for Poplar the runtime increases by roughly 50%. We observe a 2.1× improvement in terms of server WAN runtime for PLASMA compared to Poplar since PLASMA incurs significantly less communication for $\mathcal{T} = 1\%$.

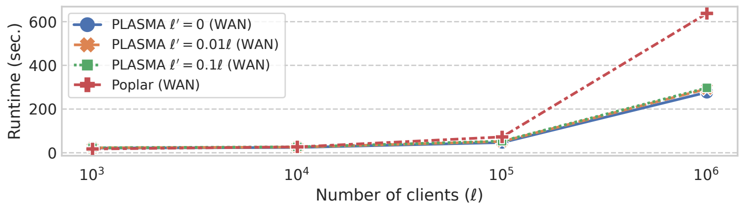


Fig. 11: Server runtime (over WAN) for an increasing number of clients (Bit string size n=64 bits).

Server-to-Server Communication. We compare the total communication costs incurred by all servers for an increasing number of clients, $\mathcal{T} = 1\%$, and $n = 256$ bit strings in Fig. 12. Poplar servers incur 35 GB of communication, whereas, PLASMA servers communicate less than 1 GB of data when considering $\ell' = 0\%, 1\%$, and 10% maliciously corrupt clients, hence yielding a 35× improvement over Poplar. The implementation of [4] is not open-source so we estimate the communication cost of [4] in Appendix G. The protocol of [4] communicates 45 GB of data to compute heavy-hitters over 10^6 client submitted 256-bit inputs. This yields a 45× improvement of PLASMA over [4].

Server Monetary Cost. To obtain a fair comparison between Poplar, [4], and PLASMA, we perform cumulative monetary cost analysis for a varying number of clients, assuming \$0.05/GB and \$1.53/hour. To estimate the monetary cost, we run PLASMA and Poplar in a similar setup as [4] and compare it with the

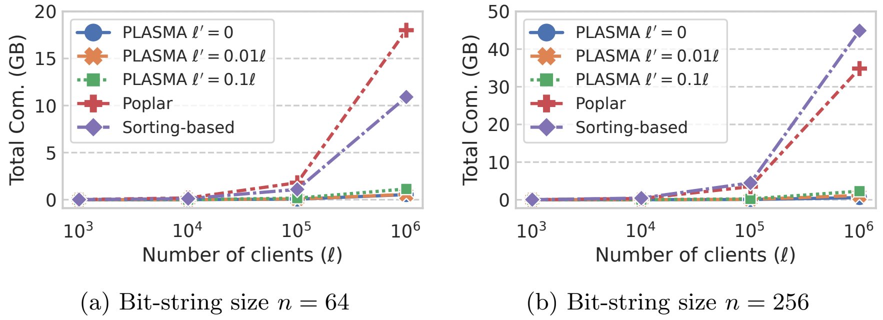


Fig. 12: Comparisons with Poplar [10] and the sorting-based approach of [4] in terms of total server-to-server communication (in GB).

runtime provided in [4]. Note that Poplar runs two servers while PLASMA runs three. The monetary cost, due to runtime, incurred by Poplar (resp. PLASMA) is two (resp. three) times the cost, due to runtime, incurred by a single Poplar (resp. PLASMA) server. After incorporating the monetary costs due to server communication, we present our findings in Fig. 13 for $\mathcal{T} = 1\%$ of the clients. Notably, for computing the $\mathcal{T}$ most popular strings among 1 million clients with $n = 256$ bit strings, Poplar costs \$4.7, while PLASMA costs \$1.78-\$1.82 for 0%-10% malicious clients, yielding a $2.5\times$ improvement over Poplar. Meanwhile, [4] costs at least \$2.24 to perform the same task, so PLASMA yields a $1.2\times$ improvement over [4] despite PLASMA having a $15\times$ runtime slowdown. This is largely due to the communication incurred by [4] for performing secure sorting under MPC. When considering input strings of smaller size, like $n = 64$, PLASMA is $3.5\times$ cheaper than Poplar and $1.4\times$ cheaper than [4].

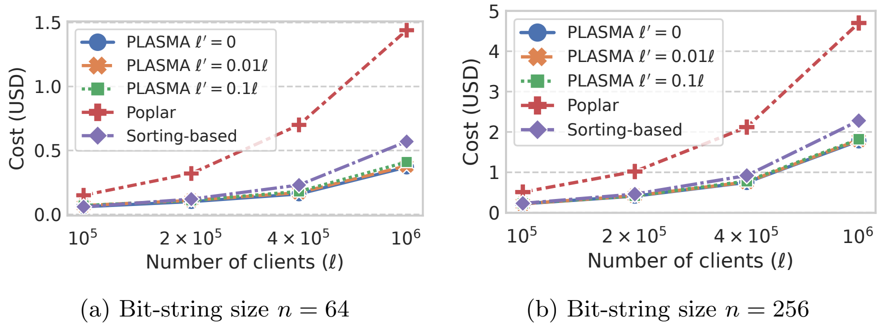


Fig. 13: Comparisons with Poplar and the sorting-based approach of [4] in terms of total monetary cost (in USD).

Applications. We discuss two realistic applications:

Popular URLs. Each URL is represented as a 256-bit string and 10000 most popular URLs are computed among 1 million client-submitted URLs, assuming $\mathcal{T} = 1\%$. Server runtimes of PLASMA and Poplar are reported in Fig. 10 (d) and the client communication costs in Figs. 9 (a) and (b) for $n = 256$. This benchmark is completed in under 22 minutes with less than 1 GB of data of communication for PLASMA, while Poplar servers incur $1.5\times$ additional runtime costs and communicate 35 GB.

Popular GPS coordinates. We employ *plus codes* [33] to efficiently encode the client GPS coordinates using 64 bits. This approach uses a grid system aligned on top of the world map, assigning specific codes to each area. 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 $\mathcal{T} = 1\%$ of the clients) among a set of client-provided inputs using 64-bit strings in roughly 4 minutes for 10^6 clients, as shown in Fig. 10 (b). Client cost is shown in Figs. 9 (a) and (b) for $n = 64$.

7 Further Extensions

We discuss two interesting extensions of PLASMA and compare them with the state-of-the-art protocol of [4]:

Fairness: The notion of fairness ensures that if an adversary receives an output then the honest parties also receive the correct output. If the adversary aborts then the honest parties also abort. In our case, we observe that the count is secret shared between the servers and based on the output of $\mathcal{F}_{\mathsf{CMP}}$ in the pruning phase, the servers compute the heavy-hitting prefix set. As a result, PLASMA is fair if the pruning phase is fair. This happens if $\mathcal{F}_{\mathsf{CMP}}$ functionality is implemented using a three-party subprotocol [15] that guarantees fairness against one malicious party. Hence, PLASMA can satisfy a stronger notion of security as compared to Poplar or [4], which only satisfies security with selective abort.

Heavy-Hitters over Multiple Thresholds: PLASMA enables computing heavy-hitters over multiple thresholds $(\mathcal{T}_1, \mathcal{T}_2, \ldots)$ based on some pre-agreed strings by the servers. This enables new applications like traffic avoidance, since different roads may have different traffic densities (e.g., highways are busier than smaller suburban roads). The servers consider that during evaluation and use higher values of $\mathcal{T}$ for highways with more vehicles and lower values for smaller roads. Conversely, it is unclear how to extend [4] to support this feature. Protocol details are in Appendix E.

8 Concluding Remarks

In this work, we presented 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, consider security against malicious clients and were prone to additive attacks by a malicious server, compromising the correctness of the protocol. To address this challenge, PLASMA introduces a novel hash-based primitive, called verifiable incremental distributed point functions, which allows the servers to validate client inputs using inexpensive operations. Additionally, we introduce a new batched consistency check that uses Merkle trees to validate multiple client sessions in a batch. This drastically reduces the concrete server-to-server communication, incurred during the heavy-hitters computation.

We implemented PLASMA using Rust and we compared it with state-of-the-art protocols like Poplar [10] and the sorting-based approach of [4]. In our experimental setup, we observe that PLASMA runs 2.1× faster than Poplar and incurs 35× and 45× less server communication than Poplar and [4], respectively. In the same conditions, PLASMA is $1.2 - 1.4\times$ cheaper than [4] and $2.5 - 3.5\times$ cheaper than Poplar respectively, corresponding to different input sizes.

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. Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA) white paper, 2021.

4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.
10. 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.
11. 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.
12. 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.
13. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.
14. Benjamin Case, Richa Jain, Alex Koshelev, Andy Leiserson, Daniel Masny, Ben Savage, Erik Taubeneck, Martin Thomson, and Taiki Yamaguchi. Interoperable Private Attribution: A Distributed Attribution and Aggregation Protocol. Cryptology ePrint Archive, Report 2023/437, 2023. https://eprint.iacr.org/2023/437.
15. Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *ACM SIGSAC CCSW@CCS 2019*, pages 81–92, London, UK, 2019. ACM.
16. 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.
17. 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.
18. Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A Private Time-Series Database from Function Secret Sharing. In *2022 IEEE Symposium on Security and Privacy*, pages 2450–2468, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.
19. Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable Distributed Aggregation Functions. *Proceedings on Privacy Enhancing Technologies*, 2023(4):578–592, July 2023.
20. 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.
21. Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*,

volume 4004 of *Lecture Notes in Computer Science*, pages 486–503, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

22. Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany.
23. Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private collection of traffic statistics for anonymous communication networks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 1068–1079, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.
24. 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.
25. 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.
26. 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.
27. 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.
28. 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. Google LLC. Open Location Code. https://github.com/google/open-location-code, 2019.
34. 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.
35. 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.
36. Antigoni Polychroniadou, Gilad Asharov, Benjamin E. Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime Match: A Privacy-Preserving Inventory Matching System, 2023.
37. 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.
38. 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 Variants of Distributed Point Functions

Incremental and Verifiable DPF (IDPF and VDPF). The IDPF [10] and VDPF [20] 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 [10], 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 [20] considers efficient hashing-based verifiable properties to ensure that a DPF (not IDPF) key is well-formed. Moreover, [20] 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.$\mathsf{Gen}(1^\kappa, f_{\alpha,\beta}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$. Given the security parameter 1^κ and a function f, output keys $\mathsf{key}_0, \mathsf{key}_1$.
– VDPF.$\mathsf{BatchEval}(b, \mathsf{key}_b, \mathbf{X}) \rightarrow (\mathbf{Y}_b, \pi_b)$: 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 $\mathbf{Y}_b \coloneqq \{y_{b,1}, y_{b,2}, \ldots, y_{b,m}\}$.

Correctness ensures that $\mathbf{Y}_0 + \mathbf{Y}_1 = f_{\alpha,\beta}(\mathbf{X})$. 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.

B Verifiable Incremental DPF

We present the verifiable incremental DPF construction, denoted as π_{VIDPF}, in Figs. 14 and 15. Our VIDPF construction is obtained by adding verifiability (steps 15-17 from Fig. 14) on top of the IDPF construction of Poplar. The security of our protocol is summarized in Theorem 2.

Theorem 2. *Assuming* $(\mathsf{PRG}, \mathsf{PRG}', \mathsf{PRG}'')$ *are pseudorandom generators, and* $(\mathrm{H}_1, \mathrm{H}_2)$ *are random oracles then* $\pi_{\mathsf{VIDPF}} = (\mathsf{Gen}, \mathsf{EvalPref})$ *in Figs. 14 and 15 is a VIDPF.*

Proof. Input privacy of our VIDPF follows from the input privacy of the underlying IDPF protocol from Poplar, which in turn relies on the pseudorandomness of PRG. Adding $\mathsf{cs}^{(i)}$ in steps 16-17 does not affect the input privacy of the client in the random oracle model since $\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$ is an `XOR` of two random oracle outputs. Each server will know the preimage of either $\widetilde{\pi}_0^{(i)}$ or the preimage of $\widetilde{\pi}_1^{(i)}$ by evaluating the given VIDPF key. The server breaks input privacy if it computes both preimages. However, to compute the other preimage it needs to invert the random oracle on $\widetilde{\pi}_{1-b'}^{(i)}$ (assuming it obtained the preimage of $\widetilde{\pi}_{b'}^{(i)}$ by evaluating the VIDPF key).

A malicious client breaks the verifiability property if there are two non-zero paths, say u and v in the evaluation tree such that the client still passes the verification check performed by the servers on $\mathsf{cs}^{(i)}$. This means the servers obtain $s_0^i(u)$, $s_1^i(u)$, $s_0^i(v)$ and $s_1^i(v)$ from Step 11 of $\mathsf{EvalNext}$ (Fig. 15) by evaluating on u and v such that the following holds:

$$s_0^i(u) \neq s_1^i(u) \text{ and } s_0^i(v) \neq s_1^i(v)$$

$$\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)}(u) \oplus \widetilde{\pi}_1^{(i)}(u) = \widetilde{\pi}_0^{(i)}(v) \oplus \widetilde{\pi}_1^{(i)}(v),$$

where $\widetilde{\pi}_b^{(i)}(u) \coloneqq \mathrm{H}_1(u, s_0^i(u))$ and $\widetilde{\pi}_b^{(i)}(v) \coloneqq \mathrm{H}_1(v, s_0^i(v))$ for $b \in \{0,1\}$. However, this is not possible in the random oracle model since it breaks the `XOR`-collision-resistance property of the random oracle H_1. The adversary cannot find such a set of $s_0^i(u)$, $s_1^i(u)$, $s_0^i(v)$ and $s_1^i(v)$ values. Lemma 3 of [20] captures the formal details. In addition, we also rely on the collision resistance property of H_2 for arguing verifiability when multiple proofs are iteratively hashed together in step 12 of the $\mathsf{EvalNext}$ algorithm. □

Notation: We denote the private n-bit string α and its bit decomposition as $\alpha_1, \ldots, \alpha_n \in \{0,1\}^n$.
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 $\mathsf{Diff} := L$, $\mathsf{Same} := R$                    ▷ Set right children to be equal.
6:      else $\mathsf{Diff} := R$, $\mathsf{Same} := L$                                  ▷ 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:     $\widetilde{\pi}_b^{(i)} = \mathrm{H}_1(\alpha_{\leq i} \| s_b^{(i)})$
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 $b \in \{0,1\}$

Convert_G(s):
1:  Let $u \leftarrow |\mathbb{G}|$.
2:  if $u = 2^m$ for an integer $m$ then:
3:      Return the group element represented by $\mathsf{PRG}'(s) \mod u$,
4:      where $\mathsf{PRG}' : \{0,1\}^\kappa \to \{0,1\}^m$.
5:  else:
6:      Let $n = \lceil \log_2 u \rceil + \kappa$.
7:      Return the group element represented by $\mathsf{PRG}''(s) \mod u$,
8:      where $\mathsf{PRG}' : \{0,1\}^\kappa \to \{0,1\}^n$.
```

Fig. 14: Protocol π_{VIDPF} for Verifiable Incremental DPF (continues in Fig. 15).

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

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

Fig. 15: Protocol π_{VIDPF} for Verifiable Incremental DPF (continuing from Fig. 14).

C Proof of Heavy-Hitters Protocol π_{HH}

C.1 Proof Sketch

Proof. The adversary is allowed to corrupt $\ell' \leq \ell$ clients and one of the servers. The other two servers are honest. 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 $p \| 0 \in \{0,1\}^{k+1}$ by constructing the VIDPF on $(p \| 0, \widetilde{\beta^k})$, i.e., $f(p \| 0) = \widetilde{\beta^k}$, where $\widetilde{\beta^k} > 1$, instead of $(p \| 0, 1)$. This is detected by the honest servers since they perform a local subtree verification by reconstructing the value $y^p - (y^{p\|0} - y^{p\|1})$ and verifying that it equals 0 for all $k > 0$. For $k = 0$, the servers verify that $y^\epsilon = 1$. Combining all k checks ensures that $y^{p\|0} = 1$ if and only if $y^p = 1$ and $y^{p\|1} = 0$, else $y^{p\|0} = 0$.
- *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 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 $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ 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 $\widehat{h^{p\|0}} := \mathrm{H}_1({y^{p\|0}_{(0,1)}}' - {y^{p\|0}_{(0,2)}}', {y^{p\|0}_{(0,2)}}' - y^{p\|0}_{(2,1)})$ and $\widehat{h^{p\|1}} := \mathrm{H}_1({y^{p\|1}_{(0,1)}}' - {y^{p\|1}_{(0,2)}}', {y^{p\|1}_{(0,2)}}' - y^{p\|1}_{(2,1)})$ where ${y^{p\|0}_{(0,1)}}', {y^{p\|0}_{(0,2)}}', {y^{p\|1}_{(0,1)}}', {y^{p\|1}_{(0,2)}}'$ are incorrect. This would allow $\mathcal{S}_0$ to introduce an additive error into the frequency for $p \,\|\, 0$ and $p \,\|\, 1$ (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 $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ sent by $\mathcal{S}_0$. This would allow the $\mathcal{S}_1$ to introduce an additive error into the frequency for $p \,\|\, 0$ and $p \,\|\, 1$ (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.

□

C.2 Formal Proof Details of Theorem 1

Security of our protocol relies on the correctness of π_{check}. π_{check} is a protocol where two honest parties commit to their inputs using Merkle-tree-based commitments and then they decommit based on whether the root commitments match or not. Correctness of π_{check} follows in a straightforward manner from the binding property of the Merkle-tree commitment, which in turn follows from the collision-resistance property of the hash function used in π_{check}.

Next, we prove the security of our protocol in the real-ideal world paradigm of *Canetti (Journal of Cryptology '00)* [13]. Let $\mathcal{A}$ denote the real-world adversary corrupting one of the servers and ℓ' clients maliciously in the real-world execution of the protocol. Let $\textsc{real}_{\mathcal{A},\pi_{\mathsf{HH}}}$ denote $\mathcal{A}$'s view after participating in the real-world execution. Let simulator Sim be the ideal-world adversary, which given access to the algorithm of $\mathcal{A}$ and functionality $\mathcal{F}_{\mathsf{HH}}$, produces the ideal world adversarial view as $\textsc{ideal}_{\mathsf{Sim},\mathcal{F}_{\mathsf{HH}}}$.

We prove that our protocol π_{HH} securely implements $\mathcal{F}_{\mathsf{HH}}$ functionality by providing an ideal world PPT simulator Sim for all PPT adversaries $\mathcal{A}$, and show that the real and ideal world view are indistinguishable, i.e., $\textsc{real}_{\mathcal{A},\pi_{\mathsf{HH}}} \overset{c}{\approx} \textsc{ideal}_{\mathsf{Sim},\mathcal{F}_{\mathsf{HH}}}$. We use a sequence of hybrids (i.e., HYB_0 - HYB_4) to prove the indistinguishability argument.

Proof. We first consider the case where $\mathcal{A}$ corrupts server $\mathcal{S}_2$ along with ℓ' clients. Then, we consider the case where $\mathcal{A}$ corrupts either $\mathcal{S}_0$ or $\mathcal{S}_1$ along with ℓ' clients.

$\mathcal{S}_2$ is corrupt. We provide the formal simulator in Fig. 16 and argue indistinguishability as follows.

- HYB_0 : The real world execution of the protocol.
- HYB_1 : Same as HYB_0, except Sim aborts if a malicious client i has provided inconsistent u_i and v_i inputs to $\mathcal{S}_0$ and $\mathcal{S}_1$ and yet passed the batched consistency check π_{check}. The two hybrids are indistinguishable due to the correctness of π_{check}.
- HYB_2 : Same as HYB_1, except the Sim extracts the corrupt client's inputs using the three pairs of DPF keys. Then Sim runs Step (iii) of simulated Batch-Verification, i.e., Sim aborts if 1) the client's input α_i is k-bits heavy-hitting, 2) $\alpha_i \parallel 0$ or $\alpha_1 \parallel 1$ is invalid, and 3) client evaded the Batch-Verification check for the sessions run between honest servers. The two hybrids are indistinguishable due to the verifiability property of VIDPF in the random oracle model. This occurs when the client successfully evades the input extraction process of VIDPF by providing malformed VIDPF keys and yet passes the batch verification checks.
- HYB_3 : Same as HYB_2, except Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the extracted inputs to obtain the $\mathsf{HH}^{\leq n}$ set and simulates $\mathcal{F}_{\mathsf{CMP}}$ based on whether a prefix γ is in $\mathsf{HH}^{\leq n}$ or not. The two hybrids are indistinguishable against a corrupt server $\mathcal{S}_2$ in the $\mathcal{F}_{\mathsf{CMP}}$-model.
- HYB_4 : Same as HYB_3, except Sim simulates the DPF key generation for the honest clients with input $(\alpha, (\beta_1, \ldots, \beta_n)) = (1, (1, \ldots, 1))$ and sets the counters to 0s in the aggregation step. Indistinguishable due to VIDPF input privacy. The 0-valued counters in the aggregation step are identically distributed to the actual aggregation counters since HYB_3 and HYB_4 are in the $\mathcal{F}_{\mathsf{CMP}}$-model. This is the ideal world execution of the protocol, completing our simulation algorithm.

Either $\mathcal{S}_0$ or $\mathcal{S}_1$ is corrupt. Next, we consider the case where either server $\mathcal{S}_0$ or $\mathcal{S}_1$ is corrupted along with ℓ' clients. We provide the simulator in Fig. 17 and argue indistinguishability as follows. (This case is similar to the case where $\mathcal{S}_1$ is corrupted along with ℓ' clients.)

- HYB_0 : The real world execution of the protocol.
- HYB_1 : Same as HYB_0, except Sim aborts if a malicious client i has provided values $(R^k_{(2,0)}, R^k_{(2,1)})$ to $\mathcal{S}_2$ and values $(R^k_{(2,0)}, R^k_{(1,2)})$ to $\mathcal{S}_1$ such that they are not equal, and yet client i passed the batched consistency check π_{check}. The two hybrids are indistinguishable due to the correctness of π_{check}.
- HYB_2 : Same as HYB_1, except Sim extracts the corrupt client's inputs following the extraction algorithm using the pair of DPF keys. Then Sim runs Step (iv) of simulated Batch-Verification, i.e., Sim aborts if 1) the client's input α_i is k-bits heavy-hitting, 2) $\alpha_i \parallel 0$ or $\alpha_1 \parallel 1$ is invalid, and 3) client evaded the Batch-Verification check for the sessions run between honest servers. The two hybrids are indistinguishable due to the verifiability property of VIDPF in the random oracle model. This occurs when a malicious client successfully evades the input extraction process of VIDPF by providing malformed VIDPF keys and yet passes the batch verification checks performed on the VIDPF proofs.
- HYB_3 : Same as HYB_2, except Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the extracted inputs to obtain $\mathsf{HH}^{\leq n}$ set and simulates the $\mathcal{F}_{\mathsf{CMP}}$ functionality based on whether a prefix γ is in $\mathsf{HH}^{\leq n}$ or not. The two hybrids are indistinguishable against a corrupt server $\mathcal{S}_0$ in the $\mathcal{F}_{\mathsf{CMP}}$-model.

Simulator Sim for maliciously corrupt ℓ' number of clients and server $\mathcal{S}_2$

– **Corruption:** Server $\mathcal{S}_2$ and ℓ' number of clients are maliciously corrupt. The rest $\ell - \ell'$ clients and servers $(\mathcal{S}_0, \mathcal{S}_1)$ are simulated by simulator Sim.

– **Primitive:** VIDPF $:= (\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

```
1: Client $\mathcal{C}$ Computation.                                        (Repeated for $\ell$ clients)
  (a) If the client is honest: Sim simulates the client by preparing three pairs of DPF keys with input 1 and output values $(1, \dots, 1)$.
      $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \dots, 1), \mathbb{G})$,   $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \dots, 1), \mathbb{G})$,
      $(\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \dots, 1), \mathbb{G})$
      Sim 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 $\mathcal{S}_2$ on behalf of the client.
  (b) If the client is corrupt: 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. (Simulator Sim initializes a list $\mathsf{L}_{ext} = \{\}$ and $\mathsf{L}_{inp} = \{\}$, and simulates $\mathcal{S}_0$ and $\mathcal{S}_1$)
  – For each corrupt client $i$, the simulator performs the following for input extraction:   (Repeated for $\ell'$ corrupt clients)
    (a) Sim extracts the corrupt client's input $(\alpha'_i, \beta'_{i,1}, \dots, \beta'_{i,n})$ from the three pairs of DPF keys - $\mathsf{key}_{(0,1)}$ and $\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(0,2)}$ and $\mathsf{key}_{(2,0)}$, and $\mathsf{key}_{(2,1)}$ and $\mathsf{key}_{(1,2)}$, provided by client $i$. If the extracted values differ, then Sim takes the necessary steps below.
    (b) If the corrupt client has not provided a valid input at level $j$, i.e., 1) $\exists j \in [n]$ s.t. $\beta'_j \neq 1$ (for the smallest $j$), or 2) the extracted inputs $\alpha'_i$ (from the three sessions) in the previous step differ in the $j^{th}$ bit, i.e., $\alpha'_{i,j}$, then Sim truncates the extracted input of client $i$ to the first $j$ bits of $\alpha_i$ as $\alpha_i := \alpha_{i,\leq j-1}$. Sim sets $\mathsf{L}^{j-1}_{\mathsf{ext}} = \mathsf{L}^{j-1}_{\mathsf{ext}} \cup \{i, j-1\}$ and updates $\mathsf{L}_{\mathsf{ext}} = \mathsf{L}_{\mathsf{ext}} \cup \mathsf{L}^{j-1}_{\mathsf{ext}}$ to denote that the $i$th client's input is valid only till level $j-1$.
    (c) Sim stores the extracted input (after necessary truncation) $\alpha_i$ for client $i$ in a list $\mathsf{L}_{\mathsf{inp}}$ as $\mathsf{L}_{\mathsf{inp}} := \mathsf{L}_{\mathsf{inp}} \cup \{i, \alpha_i\}$.
  – After running the above extraction process for all corrupt clients, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the input list $\mathsf{L}_{\mathsf{inp}}$ to obtain the output set of $\mathcal{T}$-heavy hitting prefixes as $\mathsf{HH}^{\leq n}$. The functionality $\mathcal{F}_{\mathsf{HH}}$ waits for further instructions from the ideal world adversary Sim.
  – Repeat the following steps for length of $k$ bits, where $k \in [0, \dots, n-1]$:
    (a) Initialization. For prefix $p \in \mathsf{HH}^k$, Sim initialize server $\mathcal{S}_0$'s and $\mathcal{S}_1$'s aggregation variables for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows:
        Simulated $\mathcal{S}_0$ sets $\mathsf{cnt}^\gamma_{(0,1)} := \mathsf{cnt}^\gamma_{(0,2)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$,   Simulated $\mathcal{S}_1$ sets $\mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0$.
    (b) VIDPF Evaluation. For prefix $p \in \mathsf{HH}^{\leq k}$, Sim simulates $\mathcal{S}_0$ and $\mathcal{S}_1$ by running the original protocol steps. (Repeated for $\ell$ clients)
    (c) Batch-Verification.
        i. Sim simulates $\mathcal{S}_0$ and $\mathcal{S}_1$ by computing $\mathbf{u}$ and $\mathbf{v}$ following the original steps of the protocol and Sim adds the $i$th client to the list $\mathsf{L}$ of discarded clients if $u_i \neq v_i$. If client $i$ is not detected as bad by running the original protocol steps of $\pi_{\mathsf{check}}$ on $\mathbf{u}$ and $\mathbf{v}$ then Sim aborts.
        ii. Sim runs the honest protocol steps to simulate the interaction between $\mathcal{S}_2 - \mathcal{S}_0$ and $\mathcal{S}_2 - \mathcal{S}_1$ to obtain the update list $\mathsf{L}$.
        iii. Sim aborts if $\exists$ client $i$ s.t. 1) its input is $k$-bits heavy-hitting (i.e., $\alpha_i \in \mathsf{HH}^k$), 2) $\alpha_i \parallel 0$ or $\alpha_i \parallel 1$ is not valid, i.e., $\{i, k\} \in \mathsf{L}^k_{\mathsf{ext}}$, 3) client $i$ evaded the consistency check, i.e., $i \notin \mathsf{L}$.
        If Sim did not abort then for all corrupt parties in list $\mathsf{L}$ at level $k$, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ to discard the parties from the output computation of $k+1$-bit heavy-hitting prefixes. Sim obtains an updated $\mathsf{HH}^{\leq n}$ set from $\mathcal{F}_{\mathsf{HH}}$.
    (d) Aggregation. Sim simulates this step for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: (Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)
        Simulated $\mathcal{S}_0$ sets $\mathsf{cnt}^\gamma_{(0,1)} := \mathsf{cnt}^\gamma_{(0,2)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$,   Simulated $\mathcal{S}_1$ sets $\mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0$.
    (e) Pruning. For every $(k+1)$-bit string $\gamma$, Sim simulates the pruning step as follows:
        • If $\gamma \in \mathsf{HH}^{k+1}$ then Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 1 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 1 as output to the servers, s.t. $\gamma$ is included in the list of heavy-hitting strings.
        • If $\gamma \notin \mathsf{HH}^{k+1}$ then Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 0 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 0 as output to the servers, s.t. $\gamma$ gets pruned.
          If the simulator of $\mathcal{F}_{\mathsf{CMP}}$ aborts, then Sim instructs $\mathcal{F}_{\mathsf{HH}}$ to abort at level $(\perp, k+1)$ and Sim aborts this simulated execution.
        Sim has successfully simulated the $\mathsf{HH}^{k+1}$ set. Sim repeats "Server Computation" steps (starting from Step 2b) on $k+1$ bit prefixes.
3: Output Phase. Sim outputs $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings on behalf of simulated $\mathcal{S}_0$ and $\mathcal{S}_1$, and instructs $\mathcal{F}_{\mathsf{HH}}$ to send output to the honest servers $\mathcal{S}_0$ and $\mathcal{S}_1$.
```

Fig. 16: Simulation Algorithm against malicious corruption of server $\mathcal{S}_2$ and ℓ' clients.

Simulator Sim for maliciously corrupt ℓ' number of clients and server $\mathcal{S}_0$

- **Corruption:** ℓ' number of clients and server $\mathcal{S}_0$ are maliciously corrupt. The rest $\ell - \ell'$ clients and servers $(\mathcal{S}_1, \mathcal{S}_2)$ are simulated by simulator Sim. Without loss of generality, we will assume that $\mathcal{S}_0$ is corrupt; the case where $\mathcal{S}_1$ is corrupt is symmetric.
- **Primitive:** VIDPF $:= (\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

1: **Client $\mathcal{C}$ Computation.** (**Repeated for ℓ clients**)

(a) *If the client is honest:* Sim simulates the client by preparing three pairs of DPF keys with input 1 and output values $(1, \ldots, 1)$.

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

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

Sim 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 $\mathcal{S}_2$ on behalf of the client.

(b) *If the client is corrupt:* 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.** *(Simulator Sim initializes a list $\mathsf{L}_{\mathsf{ext}} = \{\}$ and $\mathsf{L}_{\mathsf{inp}} = \{\}$, and simulates $\mathcal{S}_1$ and $\mathcal{S}_2$)*

- For each corrupt client i, the simulator performs the following for input extraction: (**Repeated for ℓ' corrupt clients**)
  - (a) Sim extracts the corrupt client's input $(\alpha'_i, \beta'_{i,1}, \ldots, \beta'_{i,n})$ from the pair of DPF keys - $\mathsf{key}_{(1,2)}$ and $\mathsf{key}_{(2,1)}$, provided by client i.
  - (b) If the corrupt client has not provided a valid input at level j, i.e., $\exists j \in [n]$ s.t. $\beta'_j \neq 1$ (for the smallest j), then Sim truncates the extracted input of client i to the first j bits of α_i as $\alpha_i := \alpha_{i,\leq j-1}$. Sim sets $\mathsf{L}^{j-1}_{\mathsf{ext}} = \mathsf{L}^{j-1}_{\mathsf{ext}} \cup \{i, j-1\}$ and updates $\mathsf{L}_{\mathsf{ext}} = \mathsf{L}_{\mathsf{ext}} \cup \mathsf{L}^{j-1}_{\mathsf{ext}}$ to denote that the ith client's input is valid only till level $j-1$.
  - (c) Sim stores the extracted input (after necessary truncation) α_i for client i in a list $\mathsf{L}_{\mathsf{inp}}$ as $\mathsf{L}_{\mathsf{inp}} := \mathsf{L}_{\mathsf{inp}} \cup \{i, \alpha_i\}$.
- After running the above extraction process for all corrupt clients, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the input list $\mathsf{L}_{\mathsf{inp}}$ to obtain the output set of $\mathcal{T}$-heavy hitting prefixes as $\mathsf{HH}^{\leq n}$. The functionality $\mathcal{F}_{\mathsf{HH}}$ waits for further instructions from the ideal world adversary Sim.
- Repeat the following steps for length of k bits, where $k \in [0, \ldots, n-1]$:
  - (a) **Initialization.** For prefix $p \in \mathsf{HH}^k$, Sim initialize server $\mathcal{S}_1$'s and $\mathcal{S}_2$'s aggregation variables for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows:

$$\text{Simulated } \mathcal{S}_1 \text{ sets } \mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0, \quad \text{Simulated } \mathcal{S}_2 \text{ sets } \mathsf{cnt}^\gamma_{(2,0)} := \mathsf{cnt}^\gamma_{(2,1)} := 0.$$

  - (b) **VIDPF Evaluation.** For prefix $p \in \mathsf{HH}^{\leq k}$, Sim simulates $\mathcal{S}_1$ and $\mathcal{S}_2$ by running the original protocol steps. (**Repeated for ℓ clients**)
  - (c) **Batch-Verification.**
    - i. Sim simulates the interaction between corrupt server $\mathcal{S}_0$ and honest server $\mathcal{S}_1$ by following the protocol steps to update list L.
    - ii. Sim simulates the interaction between corrupt server $\mathcal{S}_0$ and honest server $\mathcal{S}_2$ by following the protocol steps to update list L.
    - iii. For each client i: Sim verifies that $\mathcal{S}_2$'s version of $(R^k_{(2,0)}, R^k_{(2,1)})$ matches with $\mathcal{S}_1$'s version of $(R^k_{(2,0)}, R^k_{(1,2)})$. If they don't match then Sim adds ith client to the list L of discarded clients. If client i is not detected as bad by running the original protocol steps of π_{check} between $\mathcal{S}_1$ and $\mathcal{S}_2$ then Sim aborts.
    - iv. Sim aborts if $\exists$ client i s.t. 1) its input is k-bits heavy-hitting (i.e., $\alpha_i \in \mathsf{HH}^k$), 2) $\alpha_i \parallel 0$ or $\alpha_i \parallel 1$ is not valid, i.e., $\{i, k\} \in \mathsf{L}^k_{\mathsf{ext}}$, 3) client i evaded the consistency check, i.e., $i \notin \mathsf{L}$.

    If Sim did not abort then for all corrupt parties in list L at level k, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ to discard the parties from the output computation of $k+1$-bit heavy-hitting prefixes. Sim obtains an updated $\mathsf{HH}^{\leq n}$ set from $\mathcal{F}_{\mathsf{HH}}$.
  - (d) **Aggregation.** Sim simulates this step for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: (**Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$**)

$$\text{Simulated } \mathcal{S}_1 \text{ sets } \mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0, \quad \text{Simulated } \mathcal{S}_2 \text{ sets } \mathsf{cnt}^\gamma_{(2,0)} := \mathsf{cnt}^\gamma_{(2,1)} := 0.$$

  - (e) **Pruning.** For every $(k+1)$-bit string γ, Sim simulates the pruning step as follows:
    - If $\gamma \in \mathsf{HH}^{k+1}$ then Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 1 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 1 as output to the servers, s.t. γ is included in the list of heavy-hitting strings.
    - If $\gamma \notin \mathsf{HH}^{k+1}$ then Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 0 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 0 as output to the servers, s.t. γ gets pruned.

    If the simulator of $\mathcal{F}_{\mathsf{CMP}}$ aborts, then Sim instructs $\mathcal{F}_{\mathsf{HH}}$ to abort at level $(\perp, k+1)$ and Sim aborts this simulated execution.

  Sim has successfully simulated the HH^{k+1} set. Sim repeats *"Server Computation"* steps (starting from Step 2b) on $k+1$ bit prefixes.

3: **Output Phase.** Sim outputs $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings on behalf of simulated $\mathcal{S}_1$ and $\mathcal{S}_2$, and instructs $\mathcal{F}_{\mathsf{HH}}$ to send output to the honest servers $\mathcal{S}_0$ and $\mathcal{S}_1$.

Fig. 17: Simulation Algorithm against malicious corruption of server $\mathcal{S}_0$ and ℓ' clients.

- HYB_4 : Same as HYB_3, except Sim simulates the DPF key generation for the honest clients with input $(\alpha, (\beta_1, \ldots, \beta_n)) = (1, (1, \ldots, 1))$ and sets the counters to 0s in the aggregation step. Indistinguishable due to VIDPF input privacy. The 0-valued counters in the aggregation step are identically distributed to the actual aggregation counters since HYB_3 and HYB_4 are in the $\mathcal{F}_{\mathsf{CMP}}$-model. This is the ideal world execution of the protocol, completing our simulation algorithm.

□

D Analysis of Batched Consistency check

We recall the batched consistency check in Fig. 8. P_0 and P_1 hash their individual leaves and verify the equality of their Merkle tree roots R_0 and R_1. If the roots are equal then all the leaves are equal. Otherwise, the parties verify the equality of the left children and the right children of the root node. If the left (resp. right) children are equal across the parties then the left (resp. right) subtrees are equal. If the left (resp. right) children are different, then the parties apply the above algorithm to the left (resp. right) subtree. Proceeding this way in an iterative manner down the tree, the parties identify the malformed leaves as $\overline{\mathsf{N}}_0^{\mathsf{K}}$ and $\overline{\mathsf{N}}_1^{\mathsf{K}}$ where the two trees differ. Then they match them with their initial lists of input sets $\mathbf{u}$ and $\mathbf{v}$ to identify the indices where they differ and then store those indices in L.

π_{check} requires $\mathsf{K}+1$ rounds of communication, where $\mathsf{K} = \lceil \log_2 \ell \rceil$. Next, we demonstrate that if ℓ' out of ℓ leaves differ, then the total communication is $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ hashes. The *Root Computation* is local and *Root Verification* communicates two hashes. During *Leaf Identification*, the parties communicate 4 hashes for each unequal node. At the root layer, only the roots are different. At the next layer, both children can differ. More generally, at layer $k \in [\mathsf{K}]$, there can be at most $\min(2^k, \ell')$ unequal nodes. The total communicated hashes are as follows:

$$\begin{aligned}
&2 + 4 \times (\min(2^0, \ell') + \ldots + \min(2^{\lceil \log_2 \ell \rceil}, \ell')) \\
&= 2 + 4 \times (1 + 2 + \ldots 2^{\lceil \log_2 \ell' \rceil} + \ell' + \ell' + \ldots + \ell') \\
&\leq 2 + 4 \times (2\ell' + \ell' \times (\lceil \log_2 \ell \rceil - \lceil \log_2 \ell' \rceil)) \\
&\approx 8\ell' + 4\ell'(\log_2 \ell - \log_2 \ell') = 4\ell'(\log_2 \tfrac{\ell}{\ell'} + 2).
\end{aligned}$$

We observe that the current version of π_{check} communicates roughly $4\ell'(\log_2 \frac{\ell}{\ell'} + 2)$ hashes. This can be further optimized to $2\ell'(\log_2 \frac{\ell}{\ell'} + 2)$ where only one server communicates at each level.

E 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. 18. 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 $\mathcal{T}$.

F Compatibility with Differential Privacy

It is straightforward to complement PLASMA with ϵ-differential privacy techniques and ensure that the presence or absence of a single client does not reveal anything about their data [22]. In this case, running two instances of PLASMA, one with $\ell - 1$ clients and another just by adding client $\mathcal{C}$, should protect the private

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 $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} \coloneqq \{\epsilon, \emptyset, \ldots, \emptyset\}$, where $\mathsf{HH}^0$ contains empty string $\epsilon$ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
– Set $\mathcal{T} \coloneqq \min(\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_m)$.
– 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 $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 0\}$.
    • If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 1\}$.
– 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 $\mathsf{HH}^n \coloneqq \mathsf{HH}^n \cup (p \parallel 0)$.
    • 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 $\mathsf{HH}^n \coloneqq \mathsf{HH}^n \cup (p \parallel 1)$.
– Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.
```

Fig. 18: Algorithm for computing heavy hitters with different thresholds from $\mathcal{T}$-prefix count queries.

data of the new client from anyone observing the outputs of the two protocols. Additionally, honest clients should not be able to be identified when a malicious server attempts to ignore honest client data to infer their inputs based on the protocol output. Therefore, PLASMA is directly compatible with the well-studied techniques from [21, 23] and can adopt a similar approach as Poplar to bound the amount of information that an adversary $\mathcal{A}$ can deduce from PLASMA's output. Like Poplar, we need to ensure that the outputs of these prefix-count oracle queries are differentially private, which can be achieved by introducing noise on the oracle's output with parameter $1/\epsilon$ from a Laplace distribution.

G Communication Cost of [4]

We analyze the total server-to-server communication cost for the sorting-based protocol of [4] (considering that its implementation is not open-source). We start from the optimized semi-honest communication cost from Appendix A.3 of [4], shown below: $mn(\frac{7}{3} + \frac{32}{9}||R||) + 3m||R|| + 2m||R'||$ 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 [34, 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. [16] 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.