PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries

Dimitris Mouris* †
University of Delaware & Nillion
Newark, DE, USA
jimouris@udel.edu

Pratik Sarkar*
Supra Research
Kolkata, WB, India
pratik93@bu.edu

Nektarios Georgios Tsoutsos
University of Delaware
Newark, DE, USA
tsoutsos@udel.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

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 [16, 26, 31, 34, 40]. 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 [6, 10, 19], 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 [10, 30], to discovering commonly used passwords [39], learning new words typed by users and identifying frequently used emojis [27], to name a few. Private heavy-hitters allow computing these results while also preserving client privacy. Existing protocols (such as [2, 8, 10, 39]) only focus on the "popular" inputs and disregard other inputs that appear less than $\mathcal{T}$ times (i.e., they are pruned by the protocol). This renders private heavy hitters a suitable candidate for finding the most common client entries, such as computing crowded areas using client-provided GPS coordinates.

Different Approaches. The literature considers the setting where two or more servers collect client inputs and run the private heavy-hitters protocol. A notable approach based on differential privacy (DP) is [2] (we discuss DP-based solutions in Section 1.2). While these protocols are computationally fast, they are limited to DP-based privacy guarantees for the client. Likewise, MPC-based solutions [8] employ general-purpose secure computation frameworks (e.g., MP-SPDZ [33], 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, 32]. The underlying protocols perform secure sorting of client inputs under MPC [4, 32] and then aggregate the sorted data, guaranteeing that private inputs remain hidden when a majority of the servers are honest. However, the communication of all aforementioned solutions is linearly dependent on the number of clients, resulting in high server-to-server communication costs.

Distributed point functions (DPFs) [12] 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 incremental DPFs (IDPF), detailed in Appendix A. It provides an IDPF-based solution for private heavy-hitters in the two-server setting,

*The first two authors have equal contribution and appear in alphabetical order.
†Research mainly conducted at the University of Delaware and completed at Nillion.

Proceedings on Privacy Enhancing Technologies 2024(3), 4–24

https://doi.org/10.56553/popets-2024-0064

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 [12, 13, 29]	●	◐†	○	○	○	2+
Poplar (IDPF) [10]	●	◐†	○	●	○	2
Bucketization (DP) [2]	●	◐†	○	●	○	2-3
MPC-based [8]	◐‡	◐†	○	○	○	3
Sorting-based [4, 32]	●	●	●	●	○	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.

and their server-to-server communication depends on the input string length in semi-honest security. For security against malicious clients, the servers validate every client's input so that malformed inputs are preemptively discarded from the computation. This is referred to as *client input validation* and it prevents malicious clients from causing an abort in the protocol. To do so, Poplar requires additional checks, which cause 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. Sabre [43] uses multi-verifier MPC-in-the-head that attests to the well-formedness of DPFs but does not focus on heavy hitters. The concurrent work of Doplar also introduced a "Verifiable IDPF (VIDPF)" similar to ours, which guarantees the same security properties. However, their constructions, namely Doplar and Prio3, rely on multiple Fully-Linear Proofs (FLPs) [21] to verify that the client's input is valid, resulting in significant communication overheads. Moreover, their approach does not consider malicious servers.

Motivation. Since all aforementioned solutions incur server-to-server communication that scales linearly with the number of clients (with large concrete communication costs), they are prohibitive for most real-world applications that require millions of clients for data collection. The concrete server-to-server communication should be 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 with PLASMA, a framework for private statistics that provides security against a malicious server and malicious clients. Our contributions are as follows:

Verifiable incremental DPF (VIDPF). First, we introduce our VIDPF primitive, which builds upon incremental DPFs (IDPF) [10] and verifiable DPFs (VDPF) [22]. 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 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 and can be used to improve earlier results in [10, 20, 21]. Previous protocols solved this problem using FLPs [9, 21] or expensive sketching that involves information-theoretic MACs [10, 11, 20]. More specifically, [21] uses FLPs in each level to verify that the client's input is one-hot, resulting in significant communication overhead as each FLP entails a large proof. Conversely, our checks for one-hot vectors do not require field multiplications, only additions and hashes which allow us to batch-verify multiple inputs together.

Batched Consistency Check. Next, we introduce a novel batched consistency check that allows us to drastically reduce 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. Here, ℓ' is the number of corrupt clients who provide malformed inputs during the protocol execution and it does not need to be a priori bounded. In case $\ell' = 0$, then our servers only exchange a pair of hashes. 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 $5 - 10\times$ 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 more than $2.5\times$ and $4\times$ cheaper respectively.

Applications. We evaluate PLASMA for two applications: a) detecting frequently visited URLs, and b) identifying popular coordinates.

Popular URLs. A prominent application (discussed both in [4] and [10]) is identifying which URLs crash the clients' browsers more frequently. 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, and compute the heavy hitter URLs that caused more than 1% of client browsers to crash. We perform the task over WAN in approximately 5 minutes for 10^6 clients, while incurring less than 1 GB of server-to-server communication (less than $1 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* [35]. We compute the heavy hitter plus codes for a threshold $\mathcal{T} = 1\%$ in under 2 minutes over WAN across 10^6 clients, while incurring very minimal server-to-server communication with $0.3 in total monetary costs.

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.

DPF-based. Distributed point functions [12] offer a straightforward solution for private histograms but they fail for heavy hitters due to the blowup in key size, as the client would need to send new DPF keys for each level, resulting in $\mathcal{O}(n)$ DPF keys for n levels. This was addressed by Poplar [10], which uses two non-colluding servers and introduces the notion of IDPFs to allow efficient evaluation of strings based on prefixes by reusing the same DPF key. Poplar's threat model is robust against malicious clients but remains susceptible to additive attacks by a malicious server. Therefore, as the servers reconstruct the output, a malicious server can add arbitrary noise to the result without the honest server realizing it. The recent works of [21, 38] propose a framework for secure data aggregation and they improve the clients' consistency checks in Poplar and Prio [19]. However, their threat model does not address additive attacks from a malicious server either. Adding such security using zero-knowledge [15, 45] is interesting future work. In contrast, PLASMA provides security against both a malicious server and malicious clients by adding one additional server. Also, Poplar still leaks some information about the heavy hitter prefixes to the servers as they reconstruct the roots of the paths before they prune them. PLASMA performs a secure comparison and either keeps the node with its subtree if $\mathcal{T} > count$, or prunes the subtree.

DP-based. There is also a body of work based on local DP and randomized responses for heavy hitters [5, 41, 46]. These techniques use a single server to collect data from clients. Therefore, this method introduces a trade-off between utility and privacy, as it leaks some information about clients' private data to the server. In contrast, other methods that provide stronger privacy guarantees require at least two non-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. Note that PLASMA is compatible with DP as we describe in Appendix F.

Sorting-based. Recent works that rely on secure sorting algorithms construct private heavy-hitter protocols [4, 32] or private ad attribution measurement [16] 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. These protocols are computationally fast over LAN. However, they perform secure sorting under MPC, and as a result, they incur heavy communication overheads and their performance degrades significantly over realistic WAN networks. 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.

General MPC-based. One could use generic MPC in the honest majority [18, 28] or dishonest majority setting [33, 44] 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 [33] and SCALE-MAMBA [1] to achieve semi-honest and malicious security, but their solution suffers from high communication and slow runtime.

3-Party Computation based. Multiple customized 3-party protocols [4, 32] aim to solve the problem of heavy-hitters. These works consider a third server to exploit the faster computation guarantees in the honest majority. Using a third server is a realistic setup and it 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 [16], JP Morgan's PrimeMatch [40], NTT's heavy-hitters protocol [4], protocols for private advertisement measurement [37], Duoram [42], Sabre [43], and others. The servers are meant to run across different organizations; for example, they can be hosted by companies and non-profit organizations as mentioned in Google-Apple's Covid Exposure system [3]. Table 1 compares our work with state-of-the-art results.

2 PRELIMINARIES

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 to disproportionally influence the result or even 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. PLASMA preserves the privacy of honest clients when one of the servers is corrupt along with any number of clients.

Servers. Similarly, a malicious server may try to deviate from the protocol 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 protocol result. The semi-honest model does not protect correctness against a malicious server, which is problematic in real-world applications, like advertisement measurements [16] between two companies, where one company may benefit from reporting inflated measurements by introducing undetectable errors. Malicious security ensures that such behaviors are caught and parties are forced to behave honestly, fostering a transparent environment for computation. Poplar has this limitation while PLASMA protects *correctness.* Hence, PLASMA is *robust* against a malicious server, since it protects both correctness and privacy. Note that in all DPF-based approaches, the servers learn the heavy prefixes, which can be beneficial in some cases (e.g., for detection of a heavy-hitting web domain that contains multiple non-heavy hitting URL errors) but can also be viewed as leakage. However, PLASMA preserves the exact counts of the prefixes.

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 κ-bit string to a pseudorandom group element of additive group $\mathbb{G}$ (where $|\mathbb{G}| > \ell$). We use $:=$ for assignment, $\xleftarrow{r} \mathcal{D}$ for sampling from distribution $\mathcal{D}$, $=$ for checking equality, and $\|$ for concatenation. For histograms, wedefine a public set $\mathbf{X}$ with m n-bit strings as $\mathbf{X} := \{x_1, x_2, \ldots, x_m\}$ 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 $x_{i,\le L} := (x_{i,1}, x_{i,2}, \ldots x_{i,L})$ for $L \le n$. Let $\mathcal{S}_b$ denote the bth server, for $b \in \{0,1,2\}$; we consider $b+1 := (b+1) \mod 3$ and $b+2 := (b+2) \mod 3$. We assume ℓ clients, each denoted as C_i for $i \in [\ell]$. For an n-bit string a we represent its bit decomposition as $a_1, \ldots, a_n \in \{0,1\}$. In histograms, each client C_i has an n-bit input string $\alpha_i \in \mathbf{X}$, for $i \in [\ell]$, while $\alpha_i \in \{0,1\}^n$ in the case of heavy-hitters. We use $\alpha_{i,1}, \ldots \alpha_{i,n} \in \{0,1\}$ to denote the bit representation of the client's input α_i.

Distributed Point Functions (DPF). Function secret sharing (FSS) [12] 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 $f_{\alpha,\beta}(x) := \beta$ 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 DPF, IDPF [10] and VDPF [22] in Appendix A for completeness.

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 $\mathcal{S}_0$ and $\mathcal{S}_1$ have a small set $\mathbf{X} := \{x_1, x_2, \ldots, x_m\}$ of m n-bit strings. Each client secret shares their input $\alpha \in \mathbf{X}$ using a DPF as $(\mathsf{key}_0, \mathsf{key}_1) := \mathsf{DPF.Gen}(1^\kappa, \alpha, 1, \mathbb{G})$. The client sends key_0 to $\mathcal{S}_0$ and key_1 to $\mathcal{S}_1$. Upon receiving the client key, each server $\mathcal{S}_b$ evaluates the DPF on all m strings of $\mathbf{X}$ as $y_b := \{\mathsf{DPF.Eval}(b, \mathsf{key}_b, x_i)\}_{x_i \in \mathbf{X}}$ and computes a vector of output shares $y_b \in \mathbb{F}^m$, for some large enough finite field $\mathbb{F}$ and $m = |\mathbf{X}|$. The servers repeat this for multiple clients and aggregate the y_b vectors in a counter vector Y_b. Finally, the servers exchange Y_0 and Y_1 to compute the output histogram as $Y := Y_0 + Y_1$. 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, Poplar allows a malicious server to add an error to its shares of the output without the honest server realizing it. For instance, say $\mathcal{S}_0$ is malicious and introduces additive errors (e.g., $\delta \in \mathbb{F}^m$) in $Y_0' := Y_0 + \delta$. That way, the output Y of the histogram would be biased by δ as $Y := Y_0' + Y_1 = Y_0 + Y_1 + \delta$. The honest server $\mathcal{S}_1$ cannot 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 Basic Histogram Protocol

We address Poplar's limitations by (1) introducing one additional server, (2) building upon the primitive of verifiable DPF [22] (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.

Our work provides the first maliciously secure protocol whose server-to-server communication is logarithmic in the total number of clients ℓ. Our servers communicate $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ hashes for consistency checks, where ℓ' is the number of corrupt clients. Similar to Poplar, we ensure input validation against malicious clients (i.e., honest servers preemptively detect inconsistent inputs and discard them). 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. 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.

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

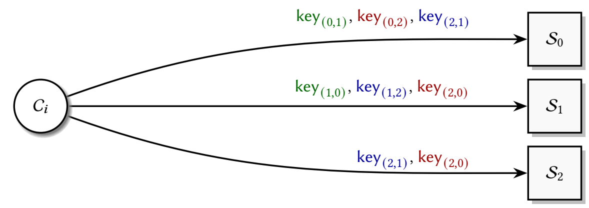


Figure 1: Distribution of session keys by client C_i.

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

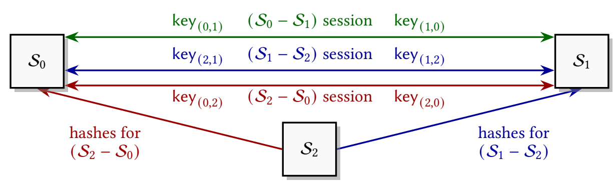


Figure 2: Session keys and attestation by $\mathcal{S}_2$.

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 [22] 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., $\mathrm{Y}_{(0,1)}, \mathrm{Y}_{(1,0)}$) 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 $h := \mathrm{H}(\mathrm{Y}_{(0,1)} - \mathrm{Y}_{(0,2)} \parallel \mathrm{Y}_{(0,2)} - \mathrm{Y}_{(2,1)})$ to $\mathcal{S}_1$, who verifies that $h = \mathrm{H}(\mathrm{Y}_{(2,0)} - \mathrm{Y}_{(1,0)} \parallel \mathrm{Y}_{(1,2)} - \mathrm{Y}_{(2,0)})$. 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, $\mathrm{Y}_{(0,1)} + \mathrm{Y}_{(1,0)} = \mathrm{Y}_{(0,2)} + \mathrm{Y}_{(2,0)} = \mathrm{Y}_{(1,2)} + \mathrm{Y}_{(2,1)}$ and thus, $\mathrm{Y}_{(0,1)} - \mathrm{Y}_{(0,2)} = \mathrm{Y}_{(2,0)} - \mathrm{Y}_{(1,0)}$ and $\mathrm{Y}_{(0,2)} - \mathrm{Y}_{(2,1)} = \mathrm{Y}_{(1,2)} - \mathrm{Y}_{(2,0)}$. 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. We present the histogram protocol in Appendix H.

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 prefix $p \in \{0,1\}^*$ over client inputs. Then, they implemented prefix count queries by relying on IDPFs (summarized in Appendix A). However, they leak 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 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 p, otherwise, it returns 0. We define it as:

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 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 k in set HH^k, which contains the list of k-bit strings that appear at least $\mathcal{T}$ times. The algorithm performs a breadth-first search of the prefix tree. It includes $k+1$ bit length strings $p \parallel 0$ in HH^{k+1} if $p \parallel 0$ 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 $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T})$. The same process is repeated for $p \parallel 1$. 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.

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., $n\ell/\mathcal{T} = n\ell/0.01\ell = 100n$).

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 $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel b, \mathcal{T}) = 1$, **then** $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \{p \parallel b\}$.
- Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.

Figure 3: Algorithm for computing $\mathcal{T}$-heavy hitters.

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:

- $\mathsf{Gen}(1^\kappa, 1^n, \alpha, (\beta^1, \beta^2, \ldots, \beta^n), \mathbb{G}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$: 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.
- $\mathsf{EvalPref}(b, \mathsf{key}_b, x, \mathsf{st}_b^{k-1}, \pi_b^{k-1}) \rightarrow (\mathsf{st}_b^k, y_b, \pi_b^k)$: Given a VIDPF key key_b and an input string $x \in \{0,1\}^k$ of length $k \leq n$ bits, this algorithm outputs an internal state st^k, secret-shared value $y_b \in \mathbb{G}$, and a proof $\pi_b^k \in \{0,1\}^*$.

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

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

where $(\mathsf{st}_0^k, y_0, \pi_0^k) :=\mathsf{EvalPref}(0, \mathsf{key}_0, x, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$ 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:

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

where $\mathsf{st}_0^{k-1}, \pi_0^{k-1}$ and $\mathsf{st}_1^{k-1}, \pi_1^{k-1}$ are obtained by recursively running the EvalPref algorithm on $k-1$ bits of z. The evaluators initialize $\mathsf{st}_0^0 := \mathsf{st}_1^0 := 0$ and $\pi_0^0 := \pi_1^0 := 0$. It also implicitly captures the requirement that $\widetilde{x} \in \{0,1\}^{k-1}$ is a prefix of $\widetilde{x} \in \{0,1\}^k$ for $k \in [n]$.

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 $\mathcal{S}_1$ 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 $p \parallel 0 \in \{0,1\}^k$ (note, the same process can be repeated for $p \parallel 1$). $\mathcal{S}_0$ and $\mathcal{S}_1$ evaluate the VIDPF keys for the three sessions on $p \parallel 0$ and obtain a secret share of the output $y^{p\parallel 0}$ and proof π. For an honest client, $y^{p\parallel 0}$ should be $\beta^k = 1$. However, a malicious client can construct malformed 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 $y^{p\parallel 0}$ 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 $p \parallel 0$ and $p \parallel 1$. The servers evaluate the VIDPF keys on the parent string p and sibling (of $p \parallel 0$) string $p \parallel 1$ to obtain secret shares of the output of y^p and $y^{p\parallel 1}$ respectively. The servers reconstruct $y^p - (y^{p\parallel 0} + y^{p\parallel 1})$ and verify that it is 0. The first check ensures that at most one of $y^{p\parallel 0}$ or $y^{p\parallel 1}$ is non-zero. Combining the two checks, we can conclude that either $(y^{p\parallel 0} = 0, y^{p\parallel 1} = 1)$ or $(y^{p\parallel 0} = 1, y^{p\parallel 1} = 0)$, since at most one child can equal 1 when the parent holds a value of 1. Iterating this for all k levels ensures that $y^{p\parallel 0} = 1$ iff $y^p = 1$ and $y^{p\parallel 1} = 0$, else $y^{p\parallel 0} = 0$. 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 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 $y^{p\parallel 0}$ is verified, the secret shares of $y^{p\parallel 0}$ are aggregated into counter $\mathsf{cnt}^{p\parallel 0}$. The servers repeat the above steps for all the clients in parallel to obtain secret shares of $\mathsf{cnt}^{p\parallel 0}$. 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 $p \parallel 0$. Similar steps are run for $p \parallel 1$. The comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [36].

Robustness Against a Malicious Server. Note that the above validation check assumes that both servers are honest. Otherwise, malicious behaviour is detected as described next. 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 Sec. 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 $\mathcal{O}(\ell)$, while when $\ell' = 0$ our communication is down to $\mathcal{O}(1)$. Formal details can be found in Section 5.

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. Note that this corruption can easily happen; if $\mathcal{A}$ has maliciously corrupted a server, then $\mathcal{A}$ can spawn multiple malicious clients. Additionally, if $\mathcal{A}$ controls a server, it can instruct $\mathcal{F}_{\mathsf{HH}}$ to discard an honest client's input. 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. We remark that $\mathcal{F}_{\mathsf{HH}}$ never leaks an honest client's inputs.

Parameters: Servers $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$. ℓ clients C_i for $i \in [\ell]$. $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$ agree on:

- 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* 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. $\mathsf{HH}^{\le n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \dots \mathsf{HH}^n\} := \{\{\epsilon\}, \emptyset, \dots, \emptyset\}$. For $k \in [0, \dots, n-1]$ and for each prefix $p \in \mathsf{HH}^k$, update $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup (p \parallel b)$ if $\sum_{i=1}^{\ell} \left|(\alpha_{i,\le k+1} = (p \parallel b))\right| \ge \mathcal{T}$, 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 $\mathsf{HH}^{\le n}$.
- *Clients* 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** $\mathcal{F}_{\mathsf{HH}}$ returns $\mathsf{HH}^{\le k}$ to $\mathcal{A}$ and the honest servers; additionally, $\mathcal{F}_{\mathsf{HH}}$ instructs the honest servers to abort by sending $\perp$.

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

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 C prepares three pairs of VIDPF keys for their private data point $\alpha \in \{0,1\}^n$, and output value $(\beta^1, \dots, \beta^n) := (1, \dots, 1)$ 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 initializes a set of sets for heavy-hitters as $\mathsf{HH}^{\le n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \dots, \mathsf{HH}^n\} := \{\{\epsilon\}, \emptyset, \dots, \emptyset\}$, where HH^0 is a set with the empty string ϵ, $\mathsf{HH}^1, \dots, \mathsf{HH}^n$ 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:

Initialization. For each k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$, the servers initialize to 0 a $\mathsf{cnt}^{p\parallel 0}$ (resp. $\mathsf{cnt}^{p\parallel 1}$) variable for each session to count the frequency of prefix $p \parallel 0$ (resp. $p \parallel 1$). Each server aggregates for each of the three sessions their additive shares of each frequency in their local cnt variables and uses them for pruning.

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 $y^0 + y^1 = 1$ (i.e., the non-zero root value is 1). If $k \ne 1$, then the servers reconstruct $y^p - (y^{p\parallel 0} + y^{p\parallel 1})$ and verify that it is 0, asserting that the parent value is propagated to the children correctly. Note that in either of the above cases, nothing is leaked about the client's input, apart from the fact that it is a valid submission (i.e., 1 at the root layer and correct propagation). 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 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 2e of Fig. 5.

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 to 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.

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

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

Client C Computation. **(Repeated for ℓ clients, each of which has their own private input α)**

(1) Client C with input α prepares three pairs DPF keys with independent randomness $u, v, w \xleftarrow{r} \{0,1\}^\kappa$, as follows:
$(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G})$, $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G})$, $(\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G})$

(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 $\mathcal{S}_2$.

Server Computation.

Each server $\mathcal{S}_b$ initializes $\mathsf{HH}_b^{\leq n} = \{\mathsf{HH}_b^0, \mathsf{HH}_b^1, \ldots \mathsf{HH}_b^n\} := \{\{\epsilon\}, \emptyset, \ldots, \emptyset\}$. Repeat the following steps for length of k bits, where $k \in [0, \ldots, n-1]$:

(1) **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:

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

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

(a) If $(p = \emptyset)$: then $\mathcal{S}_0$ sets $\mathsf{st}^\emptyset_{(0,1)} := \pi^\emptyset_{(0,1)} := \mathsf{st}^\emptyset_{(0,2)} := \pi^\emptyset_{(0,2)} := \mathsf{st}^\emptyset_{(2,1)} := \pi^\emptyset_{(2,1)} := \emptyset$, $\mathcal{S}_1$ sets $\mathsf{st}^\emptyset_{(1,2)} := \pi^\emptyset_{(1,2)} := \mathsf{st}^\emptyset_{(1,0)} := \pi^\emptyset_{(1,0)} := \mathsf{st}^\emptyset_{(2,0)} := \pi^\emptyset_{(2,0)} := \emptyset$. $\mathcal{S}_2$ 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 $\mathcal{S}_b$ retrieves the state from memory corresponding to the internal states of π_{VIDPF} for prefix p: $\mathcal{S}_0$ 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 $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$. $\mathcal{S}_1$ 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 $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$. $\mathcal{S}_2$ retrieves $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$ and $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$.

(b) 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:

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

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

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

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

(c) If $k = 1$: Servers compute the proof that the VIDPF evaluation at the root layer sums up to 1:

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

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

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

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

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

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

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

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

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

(f) *Client State Accumulation:* The servers accumulate their local state for each client session as follows:

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

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

$$\mathcal{S}_2, \mathcal{S}_1 \text{ set } R^k_{(2,0)} := \mathrm{H}_2(\|_{p\in\mathsf{HH}^k}(p, h^p_{(2,0)}, \pi^{p\parallel 0}_{(2,0)}, \pi^{p\parallel 1}_{(2,0)})), \text{ and } \mathcal{S}_2, \mathcal{S}_0 \text{ set } R^k_{(2,1)} := \mathrm{H}_2(\|_{p\in\mathsf{HH}^k}(p, h^p_{(2,1)}, \pi^{p\parallel 0}_{(2,1)}, \pi^{p\parallel 1}_{(2,1)}))$$

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

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 $\mathsf{cnt}^{p\parallel 1}$.

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 [36].

Server Computation (Continued from Fig. 5) Repeat the following steps for length of k bits, where $k \in [n]$:

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

(a) $\mathcal{S}_0$ 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: $(\text{ver}, \mathsf{L}) := \pi_{\text{check}}(\mathbf{u}, \mathbf{v})$:

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

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

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

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

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

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

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

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

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

(5) **Pruning.** For every $(k+1)$-bit string γ, the servers invoke $\mathcal{F}_{\text{CMP}}$ functionality (Fig. 7) with the additive shares of the node frequency.

$$\mathcal{S}_0 \text{ invokes } \mathcal{F}_{\text{CMP}}(\text{cnt}^\gamma_{(0,1)}, 0, \text{cnt}^\gamma_{(0,2)}, \text{cnt}^\gamma_{(2,1)}, \text{cnt}^\gamma_{(0,2)}, \mathcal{T}), \quad \mathcal{S}_1 \text{ invokes } \mathcal{F}_{\text{CMP}}(\text{cnt}^\gamma_{(1,0)}, \text{cnt}^\gamma_{(1,2)}, 0, \text{cnt}^\gamma_{(1,2)}, \text{cnt}^\gamma_{(2,0)}, \mathcal{T}), \quad \mathcal{S}_2 \text{ invokes } \mathcal{F}_{\text{CMP}}(0, \text{cnt}^\gamma_{(2,1)}, \text{cnt}^\gamma_{(2,0)}, 0, 0, \mathcal{T})$$

The servers abort if $\mathcal{F}_{\text{CMP}}$ aborts. If $\mathcal{F}_{\text{CMP}}$ outputs 1 set $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \gamma$. Otherwise, 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.

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

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

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

(1) **If** not $\mathcal{T}_0 = \mathcal{T}_1 = \mathcal{T}_2$, **then** $\mathcal{F}_{\text{CMP}}$ aborts. **Else**, set $\mathcal{T} := \mathcal{T}_0$.
(2) **If** $a = b = c = d = e$ and $a < \mathcal{T}$ **then** output 0.
(3) **If** $a = b = c = d = e$ and $a \geq \mathcal{T}$ **then** output 1.
(4) **Else**, $\mathcal{F}_{\text{CMP}}$ 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, the functionality instructs the honest servers to abort.

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

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).

Theorem 1. *Assuming VIDPF is a verifiable incremental DPF and H_1, H_2 are random oracles, $\mathcal{F}_{CMP}$ is a secure comparison functionality (Fig. 7), and H (in π_{check}) is collision-resistant, then π_{HH} (Figs. 5 and 6) implements $\mathcal{F}_{HH}$ in the (random oracle, $\mathcal{F}_{CMP}$)-model against malicious corruption of one server and $\ell' \leq \ell$ clients.*

Proof Sketch. Security of our protocol is captured in Theorem 1 and proven in Appendix C. Below we provide a security sketch. The adversary is allowed to corrupt $\ell' \leq \ell$ clients and one of the servers. The other two servers are honest. **A malicious client** attempts to inject an error and is detected in the following ways:

Client VIDPF keys are malformed. A malicious client can provide malformed VIDPF keys that are non-zero in more than one path in the tree. 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 VIDPF proofs. 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 $y^\epsilon = 1$.

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 $(\alpha_1, 1)$ for session $\mathcal{S}_0 - \mathcal{S}_1$, $(\alpha_2, 1)$ for session $\mathcal{S}_1 - \mathcal{S}_2$, and $(\alpha_3, 1)$ for session $\mathcal{S}_2 - \mathcal{S}_0$, where $\alpha_1, \alpha_2, \alpha_3 \in \{0,1\}^n$ and might be different. To ensure the input is consistent across sessions, 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 the VIDPF evaluation is the same across the sessions if and only if the checks pass, ensuring that $\alpha_1 = \alpha_2 = \alpha_3$. This is performed by computing $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ hashes.

A malicious server can collude with malicious clients. Observe that the honest clients' inputs are always hidden from the adversary due to input privacy of VIDPF. Next, a malicious server could 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 $\text{key}_{(2,1)}$ 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 can 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 allows $\mathcal{S}_0$ 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 $\mathcal{F}_{\mathsf{CMP}}$ for all three sessions. The reconstructed count will not 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}. The case where $\mathcal{S}_1$ is corrupt is symmetrical.

$\mathcal{S}_2$ *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. A malicious $\mathcal{S}_2$ can also collude with a malicious client, and the latter can provide malformed inputs in the three sessions. If this happens in the $\mathcal{S}_0 - \mathcal{S}_1$ 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. If the client provides malformed (e.g., 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 $\mathcal{S}_2$, it again gets detected since $\mathcal{S}_0$ computes the hashes $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ honestly and $\mathcal{S}_1$ verifies them honestly.

Round Complexity. Next, we analyze the round complexity of our heavy-hitters protocol. The Server computation is performed for n levels ($k \in [0, \ldots, n-1]$), where each level involves "VIDPF Evaluation", "Batch Verification", "Aggregation", and "Pruning" phases. The VIDPF evaluation and aggregation steps are performed locally by each server. Each batch-verification step requires $\lceil \log_2 \ell \rceil + 1$ rounds in the worst case (when there are malformed client inputs at each level) and a single round in the best case (when all the clients are honest). However, all verification steps for level k are performed in parallel and are batched. We further elaborate on this in Section 5. In the pruning phase, the servers run a protocol that implements $\mathcal{F}_{\mathsf{CMP}}$ for each prefix, which is performed in parallel for all prefixes at the same level. Instantiating $\mathcal{F}_{\mathsf{CMP}}$ with Rabbit [36] involves $\log_2 |\mathbb{G}|$ rounds, where the frequency count is performed over $\mathbb{G}$. Summing up, the best case round complexity of PLASMA is $n \cdot (1 + \log_2 |\mathbb{G}|)$ and the worst case round complexity is $n \cdot (\lceil \log_2 \ell \rceil + 1 + \log_2 |\mathbb{G}|)$. For benchmarking, we implement group $\mathbb{G}$ using a 64-bit ring to exploit native CPU ring optimizations.

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.

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

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

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

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

Unequal Leaf Identification: For $b \in \{0,1\}$, party P_b sets $\overline{\mathsf{N}}_b^1 := \mathsf{R}_b$ 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 $\mathsf{child}_b^{\mathsf{R}}$.
- Parties exchange $\mathsf{child}_0^{\mathsf{L}}$, $\mathsf{child}_1^{\mathsf{L}}$, $\mathsf{child}_0^{\mathsf{R}}$, and $\mathsf{child}_1^{\mathsf{R}}$, and perform the following for $b \in \{0,1\}$:

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

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

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 $\mathsf{L} := \mathsf{L} \cup \{i : \overline{\mathsf{N}}_{b,i}^{\mathsf{K}} = \mathsf{N}_{b,i}^{\mathsf{K}}\}$. Party P_b outputs $(\mathsf{ver}, \mathsf{L})$.

Figure 8: π_{check} for equality verification of ℓ strings between two parties and identification of unequal strings.

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. In case $\ell' = 0$, then our communication is a pair of hashes.

6 EXPERIMENTAL EVALUATIONS

We implement our heavy-hitters protocol π_{HH} in Rust and use the `tarpc` framework by Google for asynchronous Remote Procedure Calls (RPC).[1] 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. We used rings in PLASMA (instead of fields) since our checks rely on the security of VIDPF (i.e., XOR-collision resistant property that is provided by the random oracle). Conversely, the security of Poplar relies on a statistical check for the client's input validation. This check relies on the underlying

[1] Our code is available at https://github.com/TrustworthyComputing/plasma.

group size and needs 62 bits for the statistical failure probability to be 2^{-60} for intermediate levels; for the leaves, we use the default size of a finite field of $2\kappa = 256$ bits as mentioned in Poplar.

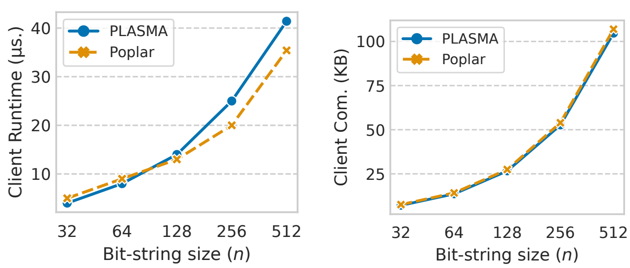
Experiment Details. Our experiments vary the number of clients between $\ell = 10^3$ and $\ell = 10^6$ with two different bit-string sizes, $n = 64$ and $n = 256$ bits. We configured the threshold $\mathcal{T}$ to be 1% of the clients' strings, and we report the client and server costs, while empirically 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[2] experiments on AWS EC2 machines (c5.9xlarge) each with 36 vCPUs at 3.60 GHz. PLASMA is compiled using Rust 1.74, 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).



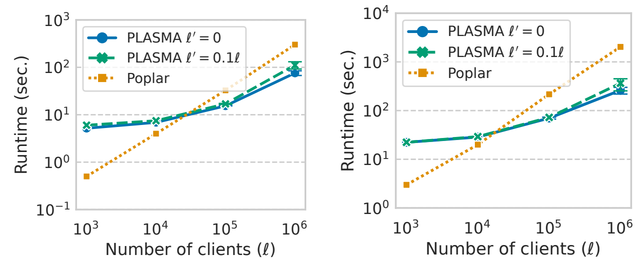
(a) Client Runtime **(b) Client Communication**

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

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

Server costs. In this experiment, we run PLASMA with randomly distributed malicious clients and compare it with Poplar. We set the malicious clients ℓ' to be a 0, 0.01, 0.1, and 0.3 fraction of the total clients ℓ. We observe that running with $\ell' = 0.01\ell$ has slightly faster performance than 0.1ℓ, while 0.3ℓ exhibits slightly worse performance than 0.1ℓ. Still, these differences are marginal compared to the total runtime,[3] so we opt for reporting the 0 and 0.1ℓ to make the figures more clear.

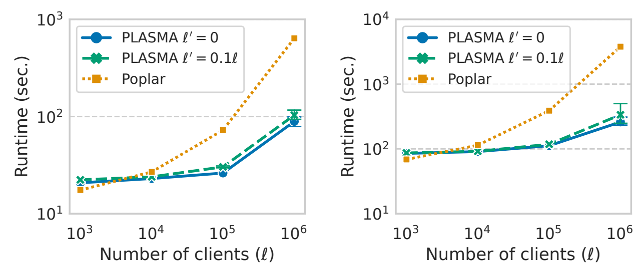
LAN Server Runtime. PLASMA outperforms Poplar in terms of server runtime by 2.7× (64 bits) and 5× (256 bits) for $\ell = 10^6$ clients and $\mathcal{T} = 1\%$ of the clients. This improvement is largely attributed to our efficient VIDPF-based client input validation. Although the presence of malicious clients has an impact on PLASMA's performance, it still remains significantly faster than Poplar as presented in Fig. 10. Meanwhile, Poplar servers validate clients' inputs using an expensive malicious secure sketching protocol.



(a) Bit-string size (n = 64) **(b) Bit-string size (n = 256)**

Figure 10: Server runtime over LAN.

WAN Server Runtime. We benchmark PLASMA and Poplar over WAN for $n = 64$ and 256 bits and 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 5-10% compared to server LAN runtime, whereas for Poplar the runtime increases by roughly 50%. We observe almost $5 - 10\times$ improvement in terms of server WAN runtime for PLASMA compared to Poplar since PLASMA incurs significantly less communication for $\mathcal{T} = 1\%$.



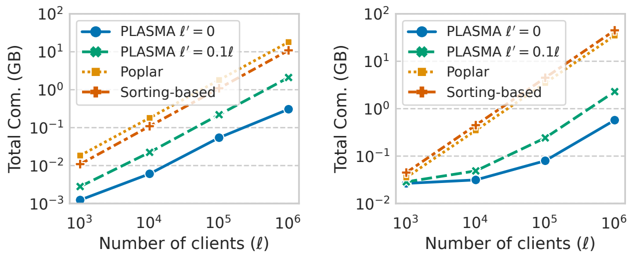
(a) Bit-string size (n = 64) **(b) Bit-string size (n = 256)**

Figure 11: Server runtime over WAN.

Server-to-Server Communication. We compare the total communication costs incurred by all servers for an increasing number of clients, $\mathcal{T} = 1\%$, and $n = 256$ in Fig. 12. Poplar servers incur 35 GB of communication, whereas, PLASMA servers communicate less than 1 GB of data when considering $\ell' = 0$ and 0.1ℓ 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].

[2]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.

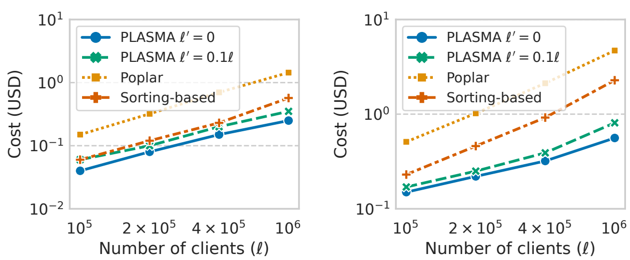
[3]Performance is impacted by expanding the Merkle tree which happens if there is at least one malicious client.



(a) Bit-string size $n = 64$ (b) Bit-string size $n = 256$

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

Server Monetary Cost. To obtain fair comparisons 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 monetary costs, we run PLASMA and Poplar in a similar setup as [4] and compare it with the runtime provided in [4, Table 7.3] (which only considers 100k-400k clients over LAN). Note that Poplar runs two servers while PLASMA runs three. The monetary cost incurred by Poplar is two times the cost incurred by a single Poplar server, while for PLASMA it's three times a single PLASMA server.



(a) Bit-string size $n = 64$ (b) Bit-string size $n = 256$

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

We present our findings in Fig. 13 for $\mathcal{T} = 1\%$ of the clients. Computing the $\mathcal{T}$ most popular strings among 1 million clients with $n = 256$ bit strings, costs \$4.7 with Poplar, while PLASMA incurs \$0.6-\$0.9 costs for 0 to 0.1ℓ malicious clients. Meanwhile, [4] costs at least \$2.2 to perform the same task, so PLASMA yields a $2.5 - 3.5\times$ improvement over [4] despite 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 $4\times$ cheaper than Poplar and $2\times$ cheaper than [4]. Lastly, Vogue [32] is not open source and it benchmarks 100k-400k clients over LAN. It claims a $6\times$ improvement over Poplar, whereas [4] claims an improvement of over $100\times$; therefore we focused on comparing with [4].

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 Figs. 10 (b) and 11 (b), while the client communication costs in Figs. 9 (a) and (b) for $n = 256$. This benchmark is completed in under 5 minutes with less than 1 GB of data of communication for PLASMA, while Poplar servers incur more than $5\times$ additional runtime costs and communicate 35 GB.

Popular GPS coordinates. We employ *plus codes* [35] 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 2 minutes for 10^6 clients, as shown in Fig. 11 (a). Client cost is shown in Fig. 9.

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 [17] 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 present PLASMA: a framework to privately identify the most popular strings – or heavy hitters – among a set of client inputs without revealing the client data points. Previous works for private heavy hitters, such as Poplar, 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.

ACKNOWLEDGMENTS

D. Mouris and N.G. Tsoutsos would like to acknowledge the support of the National Science Foundation (Award 2239334), and the Electrical & Computer Engineering department at the University of Delaware. P. Sarkar conducted the research at Boston University and he was supported by NSF Awards 1931714, 1414119, and the DARPA SIEVE program.

REFERENCES

[1] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N Smart, T Tanguy, et al. 2021. Scale–mamba v1. 12: Documentation.

[2] Erik Anderson, Melissa Chase, F. Betul Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. 2021. Aggregate Measurement via Oblivious Shuffling. Cryptology ePrint Archive, Report 2021/1490. https://eprint.iacr.org/2021/1490.

[3] Apple and Google. 2021. Exposure Notification Privacy-preserving Analytics (ENPA) white paper. , 13 pages.

[4] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 125–138. https://doi.org/10.1145/3548606.3560691

[5] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. 2017. Practical locally private heavy hitters. *Advances in Neural Information Processing Systems* 30 (2017), 1–32.

[6] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 307–321. https://doi.org/10.1145/3548606.3559383

[7] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS 2008: 13th European Symposium on Research in Computer Security (Lecture Notes in Computer Science, Vol. 5283)*, Sushil Jajodia and Javier López (Eds.). Springer, Heidelberg, Germany, Málaga, Spain, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[8] Jonas Böhler and Florian Kerschbaum. 2021. Secure Multi-party Computation of Differentially Private Heavy Hitters. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2361–2377. https://doi.org/10.1145/3460120.3484557

[9] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *Advances in Cryptology – CRYPTO 2019, Part III (Lecture Notes in Computer Science, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 67–97. https://doi.org/10.1007/978-3-030-26954-8_3

[10] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight Techniques for Private Heavy Hitters. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 762–776. https://doi.org/10.1109/SP40001.2021.00048

[11] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2023. Arithmetic Sketching. In *Advances in Cryptology – CRYPTO 2023, Part I (Lecture Notes in Computer Science)*. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 171–202. https://doi.org/10.1007/978-3-031-38557-5_6

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *Advances in Cryptology – EUROCRYPT 2015, Part II (Lecture Notes in Computer Science, Vol. 9057)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Heidelberg, Germany, Sofia, Bulgaria, 337–367. https://doi.org/10.1007/978-3-662-46803-6_12

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1292–1303. https://doi.org/10.1145/2976749.2978429

[14] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (Jan. 2000), 143–202. https://doi.org/10.1007/s001459910006

[15] Ran Canetti, Pratik Sarkar, and Xiao Wang. 2022. Triply Adaptive UC NIZK. In *Advances in Cryptology – ASIACRYPT 2022, Part II (Lecture Notes in Computer Science, Vol. 13792)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, Heidelberg, Germany, Taipei, Taiwan, 466–495. https://doi.org/10.1007/978-3-031-22966-4_16

[16] Benjamin Case, Richa Jain, Alex Koshelev, Andy Leiserson, Daniel Masny, Ben Savage, Erik Taubeneck, Martin Thomson, and Taiki Yamaguchi. 2023. Interoperable Private Attribution: A Distributed Attribution and Aggregation Protocol. Cryptology ePrint Archive, Report 2023/437. https://eprint.iacr.org/2023/437.

[17] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM SIGSAC CCSW@CCS 2019*. ACM, London, UK, 81–92.

[18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology – CRYPTO 2018, Part III (Lecture Notes in Computer Science, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 34–64. https://doi.org/10.1007/978-3-319-96878-0_2

[19] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 259–282.

[20] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A Private Time-Series Database from Function Secret Sharing. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 2450–2468. https://doi.org/10.1109/SP46214.2022.9833611

[21] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. 2023. Verifiable Distributed Aggregation Functions. *Proceedings on Privacy Enhancing Technologies* 2023, 4 (July 2023), 578–592. https://doi.org/10.56553/popets-2023-0126

[22] Leo de Castro and Antigoni Polychroniadou. 2022. Lightweight, Maliciously Secure Verifiable Function Secret Sharing. In *Advances in Cryptology – EUROCRYPT 2022, Part I (Lecture Notes in Computer Science, Vol. 13275)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, Germany, Trondheim, Norway, 150–179. https://doi.org/10.1007/978-3-031-06944-4_6

[23] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Advances in Cryptology – EUROCRYPT 2006 (Lecture Notes in Computer Science, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, Heidelberg, Germany, St. Petersburg, Russia, 486–503. https://doi.org/10.1007/11761679_29

[24] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC 2006: 3rd Theory of Cryptography Conference (Lecture Notes in Computer Science, Vol. 3876)*, Shai Halevi and Tal Rabin (Eds.). Springer, Heidelberg, Germany, New York, NY, USA, 265–284. https://doi.org/10.1007/11681878_14

[25] Tariq Elahi, George Danezis, and Ian Goldberg. 2014. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *ACM CCS 2014: 21st Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 1068–1079. https://doi.org/10.1145/2660267.2660280

[26] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *ACM CCS 2014: 21st Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 1054–1067. https://doi.org/10.1145/2660267.2660348

[27] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. 2016. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *Proc. Priv. Enhancing Technol.* 2016, 3 (2016), 41–61.

[28] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *Advances in Cryptology – EUROCRYPT 2017, Part II (Lecture Notes in Computer Science, Vol. 10211)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, Germany, Paris, France, 225–255. https://doi.org/10.1007/978-3-319-56614-6_8

[29] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014 (Lecture Notes in Computer Science, Vol. 8441)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer, Heidelberg, Germany, Copenhagen, Denmark, 640–658. https://doi.org/10.1007/978-3-642-55220-5_35

[30] Justin Hsu, Sanjeev Khanna, and Aaron Roth. 2012. Distributed Private Heavy Hitters. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I* (Warwick, UK) *(ICALP'12)*. Springer-Verlag, Berlin, Heidelberg, 461–472. https://doi.org/10.1007/978-3-642-31594-7_39

[31] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *EuroS&P*. IEEE, Genoa, Italy, 370–389.

[32] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2022. Vogue: Faster Computation of Private Heavy Hitters. Cryptology ePrint Archive, Report 2022/1561. https://eprint.iacr.org/2022/1561.

[33] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[34] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. 2021. Private Join and Compute from PIR with Default. In *Advances in Cryptology – ASIACRYPT 2021, Part II (Lecture Notes in Computer Science, Vol. 13091)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, Heidelberg, Germany, Singapore, 605–634. https://doi.org/10.1007/978-3-030-92075-3_21

[35] Google LLC. 2019. Open Location Code. https://github.com/google/open-location-code.

[36] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. 2021. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In *FC 2021: 25th International Conference on Financial Cryptography and Data Security, Part I (Lecture Notes in Computer Science, Vol. 12674)*, Nikita Borisov and Claudia Díaz (Eds.). Springer, Heidelberg, Germany, Virtual Event, 249–270. https://doi.org/10.1007/978-3-662-64322-8_12

[37] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M Case. 2024. Delegated Private Matching for Compute. *Proceedings on Privacy Enhancing Technologies* 2024, 2 (July 2024), 1–24.

[38] Dimitris Mouris, Christopher Patton, Hannah Davis, Pratik Sarkar, and Nektarios Georgios Tsoutsos. 2024. Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics. Cryptology ePrint Archive, Paper 2024/221. https://eprint.iacr.org/2024/221 https://eprint.iacr.org/2024/221.

[39] Moni Naor, Benny Pinkas, and Eyal Ronen. 2019. How to (not) Share a Password: Privacy Preserving Protocols for Finding Heavy Hitters with Adversarial Behavior. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 1369–1386. https://doi.org/10.1145/3319535.3363204

[40] Antigoni Polychroniadou, Gilad Asharov, Benjamin E. Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. 2023. Prime Match: A Privacy-Preserving Inventory Matching System. , 400 pages.

[41] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. 2016. Heavy Hitter Estimation over Set-Valued Data with Local Differential Privacy. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 192–203. https://doi.org/10.1145/2976749.2978409

[42] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. 2023. Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3907–3924. https://www.usenix.org/conference/usenixsecurity23/presentation/vadapalli

[43] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. 2022. Sabre: Sender-Anonymous Messaging with Fast Audits. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1953–1970. https://doi.org/10.1109/SP46214.2022.9833601

[44] Yongqin Wang, Pratik Sarkar, Nishat Koti, Arpita Patra, and Murali Annavaram. 2023. CompactTag: Minimizing Computation Overheads in Actively-Secure MPC for Deep Neural Networks. *IACR Cryptol. ePrint Arch.* (2023), 1729. https://eprint.iacr.org/2023/1729

[45] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2986–3001. https://doi.org/10.1145/3460120.3484556

[46] Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. 2020. Federated Heavy Hitters Discovery with Differential Privacy. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 108)*, Silvia Chiappa and Roberto Calandra (Eds.). PMLR, Online, 3837–3847. https://proceedings.mlr.press/v108/zhu20a.html

APPENDIX

A VARIANTS OF DISTRIBUTED POINT FUNCTIONS

Incremental and Verifiable DPF (IDPF and VDPF). The IDPF [10] and VDPF [22] 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 [22] considers efficient hashing-based verifiable properties to ensure that a DPF (not IDPF) key is well-formed. Moreover, [22] 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:

- $\mathsf{VDPF.Gen}(1^\kappa, f_{\alpha,\beta}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$. Given the security parameter 1^κ and a function f, output keys $\mathsf{key}_0, \mathsf{key}_1$.
- $\mathsf{VDPF.BatchEval}(b, \mathsf{key}_b, \mathrm{X}) \rightarrow (\mathrm{Y}_b, \pi_b)$: For $b \in \{0,1\}$, batch verifiable evaluation takes a set $\mathrm{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$, where each $x_i \in \{0,1\}^n$. Outputs $\mathrm{Y}_b \coloneqq \{y_{b,1}, y_{b,2}, \ldots, y_{b,m}\}$.

Correctness ensures that $\mathrm{Y}_0 + \mathrm{Y}_1 = f_{\alpha,\beta}(\mathrm{X})$. 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 the same if and only if they have been generated from valid keys key_0 and key_1 of a point function.

B VERIFIABLE INCREMENTAL DPF

We present the VIDPF 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. We have underlined the lines that focus on verifiability in these two figures. The Convert takes the corrected seed $\tilde{s}_b^{(i)}$ for level i, runs PRG$''$ and outputs κ bit seed $s_b^{(i)}$ for level i and value $W_b^{(i)}$. This occurs at the intermediate levels and is performed by executing the "else" part of Convert. $W_b^{(i)}$ comes from $\mathbb{G}$ since it generates the output W_{cw} based on intermediate β_i. At the leaves, the "if" part of Convert is executed where only $W_b^{(i)}$ is generated. The security of our protocol is summarized in Theorem 2.

Theorem 2. *Assuming (PRG, PRG', PRG'') are pseudorandom generators, PRG is κ-collision resistant and (H_1, H_2) are random oracles then $\pi_{VIDPF} = (Gen, EvalPref)$ in Figs. 14 and 15 is a VIDPF.*

We define κ-collision resistant PRG as follows:

Definition 2 (κ-Collision Resistant PRG). *We say that a $PRG : \{0,1\}^\kappa \rightarrow \{0,1\}^{2\kappa+2}$ is κ-collision resistant if a PPT adversary cannot output s_0 and s_1 such that*

$$(A_0 \parallel T_0 \parallel B_0 \parallel T_0') \coloneqq PRG(s_0),$$
$$(A_1 \parallel T_1 \parallel B_1 \parallel T_1') \coloneqq PRG(s_1),$$
$$\text{and } B_0 = B_1,$$

where $A_0, A_1, B_0, B_1 \in \{0,1\}^\kappa$ and $T_0, T_0', T_1, T_1' \in \{0,1\}$.

We recall the notion of XOR-collision resistance from [22] below for our security proof.

Definition 3 (XOR-Collision Resistance [22]). *We say a function family $\mathcal{F}$ is XOR-collision resistant if no PPT adversary given a randomly sampled $f \in \mathcal{F}$ can find four values $x_0, x_1, x_2, x_3 \in \{0,1\}^\kappa$ such that $(x_0, x_1) \neq (x_2, x_3)$, $(x_0, x_1) \neq (x_3, x_2)$, and $f(x_0) \oplus f(x_1) = f(x_2) \oplus f(x_3) \neq 0$, except with negligible probability in security parameter κ.*

It can be implemented by assuming the function f is a random oracle. Next, we proceed to the proof of Thm. 2.

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 $\mathrm{cs}^{(i)}$ in steps 16-17 (Fig. 14) does not affect the input privacy of the client in the random oracle model since $\mathrm{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$ 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 $\mathrm{cs}^{(i)}$ for $i \in [n]$. We prove this via two steps:

- *At most one non-zero value at each level i:* We prove this via contradiction. Assume a client generates VIDPF keys that evaluate to two non-zero values at level i. It 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 EvalNext (Fig. 15) by evaluating on u and v such that the following holds:
$$s_0^i(u) \neq s_1^i(u) \text{ and } s_0^i(v) \neq s_1^i(v)$$
$$\mathrm{cs}^{(i)} = \widetilde{\pi}_0^{(i)}(u) \oplus \widetilde{\pi}_1^{(i)}(u) = \widetilde{\pi}_0^{(i)}(v) \oplus \widetilde{\pi}_1^{(i)}(v),$$
where $\widetilde{\pi}_b^{(i)}(u) := \mathrm{H}_1(u, s_0^i(u))$ and $\widetilde{\pi}_b^{(i)}(v) := \mathrm{H}_1(v, s_0^i(v))$ 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 [22] captures the formal details of the reduction. In addition, the servers also check multiple proofs by iteratively hashing them together using H_2 in step 12 of the EvalNext algorithm. So, we also rely on the collision resistance property of H_2 to argue that it suffices to check the equality of the hash values computed using H_2 to ensure that the preimages are equal.
- *Non-zero value at level i+1 is a child of non-zero value at level i:* We prove this via contradiction. Assume a client generates VIDPF keys that evaluate to a non-zero value at level i on prefix $u \in \{0,1\}^i$ and a non-zero value at level $i+1$ on prefix $v \in \{0,1\}^{i+1}$ such that the non-zero node at level i is not the parent of the non-zero value at level $i+1$, i.e., $u \neq v^{\leq i}$. This means that $s_0^i(v) = s_1^1(v)$ and $s_0^i(u) \neq s_1^1(u)$ since there can be at most one pair of non-zero s values at each level. Next, consider the inputs to the EvalNext algorithms for evaluation on input prefix v in Fig. 15. We consider the following two cases:
  - *st^i is same for both servers:* In this case both $s_0^i(v) = s_1^1(v)$ and $t_0^i(v) = t_1^1(v)$. Here the input of the server to EvalNext is the same except for the value b. Hence, the evaluation algorithm of the servers on prefix v will be identical except in step 10 where server b obtains $y_b^{(i+1)}$ values such that $y_0^{(i+1)} + y_1^{(i+1)} = 0$. So, the output cannot be non-zero in this case.
  - *st^i is different for both servers:* In this case, $s_0^i(v) = s_1^1(v)$ but $t_0^i(v) \neq t_1^i(v)$. For this to happen there exists $s_0^{i-1}(v)$, $t_0^{i-1}(v)$, $s_1^{i-1}(v)$, $t_1^{i-1}(v)$ such that $(s_0^i(v), t_0^i(v))$ and $(s_1^i(v), t_1^1(v))$ are obtained by computing PRG on $s_0^{i-1}(v)$ and $s_1^{i-1}(v)$ respectively and applying Step 4 of EvalNext based on $t_0^{i-1}(v)$ and $t_1^{i-1}(v)$ values.
    - *If $v^{\leq i-1} = u^{\leq i-1}$:* then $s_0^{i-1}(v) = s_0^{i-1}(u)$ and $s_1^{i-1}(v) = s_1^{i-1}(u)$. But we know that $s_0^i(u) \neq s_1^i(u)$. We also know that $s_b^i(u)$ and $s_b^i(v)$ is generated from the same state st_b^{i-1} by server b where only $u_i \neq v_i$, which is x_i in EvalNext. In this case, steps 1-5 are the same for the evaluation of u_i and v_i. Assume $u_i = 0$ and $v_1 = 1$ without loss of generality. This means that $s_0^L \neq s_1^L$ and $s_0^R = s_1^R$, where s_b^L and s_b^R are computed by server b by evaluating the PRG on s_b^{i-1} and then XORing the output with $t^{(i-1)} \cdot s_{\mathrm{cw}}$. This breaks the collision resistance of PRG since $s_0^{i-1} \neq s_1^{i-1}$ but $(A_0 \| T_0 \| \mathbf{B} \| T_0') := \mathrm{PRG}(s_0^{i-1})$ and $(A_0 \| T_0 \| \mathbf{B} \| T_0') := \mathrm{PRG}(s_0^{i-1})$ where $A_0, A_1, B, B \in \{0,1\}^\kappa$ and $T_0, T_0', T_1, T_1' \in \{0,1\}$.
    - *If $v^{\leq i-1} \neq u^{\leq i-1}$:* then $s_0^{i-1}(v) = s_1^{i-1}(v)$ and $t_0^{i-1}(v) \neq t_1^{i-1}(v)$ and we apply our argument recursively for $i-2$ and so on until we get the previous case where $u^\ell = v^\ell$ for $\ell \in [i-1]$.

□

We note that we do not need collision resistance from the PRG since we do not require that the non-zero values lie on the same path. We only need that each level contains a single non-zero node and for that the XOR-collision resistance property suffices. This property is implemented by assuming that $(\mathrm{H}_1, \mathrm{H}_2)$ are random oracles.

C HEAVY-HITTERS PROTOCOL π_{HH} PROOF

In this section, we formally prove 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 straightforwardly 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 [14]. 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

Notation: We denote the private n-bit string α and its bit decomposition as $\alpha_1, \ldots, \alpha_n \in \{0,1\}^n$.
Primitives: $\mathrm{PRG} : \{0,1\}^\kappa \rightarrow \{0,1\}^{2\kappa+2}$ is a pseudorandom generator.
$\mathrm{H}_1 : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow \{0,1\}^{2\kappa}$ and $\mathrm{H}_2 : \{0,1\}^{2\kappa} \rightarrow \{0,1\}^{2\kappa}$ are random oracles.

Gen$(1^\kappa, 1^n, \alpha, (\beta_1, \beta_2, \ldots \beta_n), \mathbb{G})$: ▷ Generate DPF keys.
Sample $s_b^{(0)} \xleftarrow{r} \{0,1\}^\kappa$ for $b \in \{0,1\}$ ▷ Secret seeds.
Let $t_0^{(0)} := 0$ and $t_1^{(0)} := 1$
for $i := 1$ to n **do** ▷ For each bit of α.
  $s_b^L \parallel t_b^L \parallel s_b^R \parallel t_b^R := \mathrm{PRG}(s_b^{(i-1)})$ for $b \in \{0,1\}$ ▷ Parse the output of PRG as a sequence of $(\kappa \parallel 1 \parallel \kappa \parallel 1)$ bits.
  if $\alpha_i = 0$ **then** Diff $:= L$, Same $:= R$ ▷ Set right children to be equal.
  else Diff $:= R$, Same $:= L$ ▷ Set left children to be equal.
  $s_{\mathrm{cw}} := s_0^{\mathrm{Same}} \oplus s_1^{\mathrm{Same}}$
  $t_{\mathrm{cw}}^L := t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$ ▷ Left control bits not equal if $\alpha_i = 0$.
  $t_{\mathrm{cw}}^R := t_0^R \oplus t_1^R \oplus \alpha_i$ ▷ Right control bits not equal if $\alpha_i = 1$.
  $\tilde{s}_b^{(i)} := s_b^{\mathrm{Diff}} \oplus t_b^{(i-1)} \cdot s_{\mathrm{cw}}$ for $b \in \{0,1\}$ ▷ Correction.
  $t_b^{(i)} := t_b^{\mathrm{Diff}} \oplus t_b^{(i-1)} \cdot t_{\mathrm{cw}}^{\mathrm{Diff}}$ for $b \in \{0,1\}$ ▷ Correction.
  $s_b^{(i)} \parallel W_b^{(i)} := \mathrm{Convert}(\tilde{s}_b^{(i)})$ for $b \in \{0,1\}$
  $W_{\mathrm{cw}}^{(i)} := (-1)^{t_1^{(i)}} \cdot [\beta_i - W_0^{(i)} + W_1^{(i)}]$ ▷ Output correction.
  $\mathrm{cw}^{(i)} := s_{\mathrm{cw}} \parallel t_{\mathrm{cw}}^L \parallel t_{\mathrm{cw}}^R \parallel W_{\mathrm{cw}}^{(i)}$ ▷ Correction word for level i.
  $\widetilde{\pi}_b^{(i)} = \mathrm{H}_1(\alpha_{\leq i} \parallel s_b^{(i)})$
  $\mathrm{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$.
$\mathrm{key}_b := (s_b^{(0)} \parallel \mathrm{cw}^{(1)} \parallel \ldots \parallel \mathrm{cw}^{(n)} \parallel \mathrm{cs}^{(1)} \parallel \ldots \parallel \mathrm{cs}^{(n)})$ for $b \in \{0,1\}$ ▷ Key for party b.
return key_b for $b \in \{0,1\}$

Convert$_{\mathbb{G}}(s)$:
Let $u \leftarrow |\mathbb{G}|$.
if $u = 2^m$ for an integer m **then:**
  Return the group element represented by $\mathrm{PRG}'(s) \mod u$,
  where $\mathrm{PRG}' : \{0,1\}^\kappa \rightarrow \{0,1\}^m$.
else:
  Let $n = \lceil \log_2 u \rceil + \kappa$.
  Return the group element represented by $\mathrm{PRG}''(s) \mod u$,
  where $\mathrm{PRG}'' : \{0,1\}^\kappa \rightarrow \{0,1\}^n$.

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

EvalNext$(b, i, \mathrm{st}^{(i-1)}, \mathrm{cw}^{(i)}, \mathrm{cs}^{(i)}, x^{\leq i}, \pi)$: ▷ Evaluate x_i.
Parse $\mathrm{st}^{(i-1)}$ as $(s^{i-1} \parallel t^{i-1})$.
$s_{\mathrm{cw}} \parallel t_{\mathrm{cw}}^L \parallel t_{\mathrm{cw}}^R \parallel W_{\mathrm{cw}}^{(i)} := \mathrm{cw}^i$ ▷ Parse correction word.
$\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R := \mathrm{PRG}(s^{(i-1)})$ ▷ Parse the output of PRG as a sequence of $(\kappa \parallel 1 \parallel \kappa \parallel 1)$ bits.
$\tau^{(i)} := (\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R) \oplus (t^{(i-1)} \cdot [s_{\mathrm{cw}} \parallel t_{\mathrm{cw}}^L \parallel s_{\mathrm{cw}} \parallel t_{\mathrm{cw}}^R])$
$s^L \parallel t^L \parallel s^R \parallel t^R := \tau^{(i)}$ ▷ Parse $\tau^{(i)}$.
if $x_i = 0$ **then** $\tilde{s}^{(i)} := s^L, \quad t^{(i)} := t^L$ ▷ Keep left path.
else $\tilde{s}^{(i)} := s^R, \quad t^{(i)} := t^R$ ▷ Keep right path.
$s^{(i)} \parallel W^{(i)} := \mathrm{Convert}(\tilde{s}^{(i)})$ ▷ New seed and output for level i.
$\mathrm{st}^{(i)} := s^{(i)} \parallel t^{(i)}$ ▷ Save the state.
$y^{(i)} := (-1)^b \cdot [W^{(i)} + t^{(i)} \cdot W_{\mathrm{cw}}]$ ▷ Compute output at level i.
$\widetilde{\pi}^{(i)} = \mathrm{H}_1(x^{\leq i} \parallel s^{(i)})$.
$\pi = \pi \oplus \mathrm{H}_2(\pi \oplus (\widetilde{\pi}^{(i)} \oplus t^{(i)} \cdot \mathrm{cs}^{(i)}))$.
return $(\mathrm{st}^{(i)}, y^{(i)}, \pi)$

EvalPref$(b, \mathrm{key}, x \in \{0,1\}^n, \mathrm{st}^{(d-1)}, d, \pi)$: ▷ Evaluate one public bitstring x on all its bits x_i for $i \in [n]$.
Parse key as $s^{(0)} \parallel \mathrm{cw}^{(1)} \parallel \ldots \parallel \mathrm{cw}^{(n)} \parallel \mathrm{cs}^{(1)} \parallel \ldots \parallel \mathrm{cs}^{(n)}$. ▷ Parse key for party b.
if $d \neq 1$ **then** parse $\mathrm{st}^{(d-1)}$ as $(s^{(d-1)} \parallel t^{(d-1)})$,
else $t^{(0)} := b, \quad \mathrm{st}^{(0)} := s^{(0)} \parallel t^{(0)}$.
for $i := d$ to n **do** ▷ For each bit of x.
  $(\mathrm{st}^{(i)}, y^{(i)}, \pi) := \mathsf{EvalNext}(b, i, \mathrm{st}^{(i-1)}, \mathrm{cw}^i, \mathrm{cs}^i, x^{\leq i}, \pi)$.
return $(\mathrm{st}^{(n)}, y^{(n)}, \pi)$

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

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 indistinguishability.

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.

- $\mathbf{HYB}_0$: The real world execution of the protocol.
- $\mathbf{HYB}_1$: 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}.
- $\mathbf{HYB}_2$: Same as HYB_1, except the Sim extracts the corrupt client's inputs using the three pairs of DPF keys. Then Sim runs Step 3c 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.
- $\mathbf{HYB}_3$: 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.
- $\mathbf{HYB}_4$: 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.

Either $\mathcal{S}_0$ *or* $\mathcal{S}_1$ *is corrupt.* Next, we consider the case where either $\mathcal{S}_0$ or $\mathcal{S}_1$ is corrupt 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 corrupt along with ℓ' clients.)

- $\mathbf{HYB}_0$: The real world execution of the protocol.
- $\mathbf{HYB}_1$: Same as HYB_0, except Sim aborts if a malicious client i 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

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. $\mathsf{H}_1, \mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

Client C Computation. **(Repeated for ℓ clients)**

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

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

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.

(2) *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 $\mathcal{S}_2$.

Server Computation. *(Simulator Sim initializes a list $L_{ext} = \{\}$ and $L_{inp} = \{\}$, and simulates $\mathcal{S}_0$ and $\mathcal{S}_1$)*

For each corrupt client i, the simulator performs the following for input extraction: **(Repeated for ℓ' corrupt clients)**

(1) 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.

(2) 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 α'_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 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 $j-1$.

(3) 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 $k \in [0, \dots, n-1]$:

(1) **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:

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

(2) **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 ℓ clients)**

(3) **Batch-Verification.**

(a) 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 ith client to the list 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 π_{check} on $\mathbf{u}$ and $\mathbf{v}$ then Sim aborts.

(b) 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 L.

(c) 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 $\mathcal{F}_{\mathsf{HH}}$.

(4) **Aggregation.** Sim simulates this step for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: **(Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)**

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

(5) **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.

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 $\mathcal{S}_1$.

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

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 3d 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.

HYB$_4$: Same as HYB_3, except Sim simulates the key generation for the honest clients with $(\alpha, (\beta_1, \dots, \beta_n)) = (1, (1, \dots, 1))$ as input 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 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,

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

Client C Computation. **(Repeated for ℓ clients)**

(1) *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.

(2) *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 $\mathcal{S}_2$.

Server Computation. *(Simulator Sim initializes a list $L_{ext} = \{\}$ and $L_{inp} = \{\}$, 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)**

(1) 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.

(2) 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 $j-1$.

(3) 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 $k \in [0, \ldots, n-1]$:

(1) **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:

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

(2) **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)**

(3) **Batch-Verification.**

(a) 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.

(b) 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.

(c) 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.

(d) 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 $\mathcal{F}_{\mathsf{HH}}$.

(4) **Aggregation.** Sim simulates this step for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: **(Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)**

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

(5) **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.

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 $\mathcal{S}_1$.

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

the parties verify the equality of the left 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 iteratively down the tree, the parties identify the malformed leaves as $\overline{\mathsf{N}}^{\mathsf{K}}_0$ and $\overline{\mathsf{N}}^{\mathsf{K}}_1$ 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

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

Parameters: Threshold $\mathcal{T}_i \in \mathbb{N}$, for string $x_i \in \mathbf{X}$ where $|\mathbf{X}| = m$, and string length $n \in \mathbb{N}$.

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

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

Algorithm:

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

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

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 SUPPORTING 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 [24]. In this case, running two instances of PLASMA, one with $\ell - 1$ clients and another just by adding client C, should protect the private 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 [23, 25] 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 ASHAROV ET AL. [4]

We analyze the total server-to-server communication cost for the sorting-based protocol of Asharov et al. [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 [36, 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.

Asharov et al. [4] considers the compiler of Chida *et al.* [18] that converts a semi-honest protocol to a malicious protocol. However, this results in increased communication cost (i.e., 2× 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:

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

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.

H PRIVATE HISTOGRAM PROTOCOL

We present our histogram protocol π_{HIST} in Fig. 19 for the sake of completeness. The histogram protocol is a building block for our heavy-hitters protocol and *is not* our final protocol. It suffers from the limitation that the client's input should lie in the subset $\mathbf{X}$ that the servers evaluate, i.e., $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$. This leaks whether the client's input lies in $\mathbf{X}$ or not based on whether the evaluated DPF output in the consistency check is 0 or not. This issue can be addressed by using techniques from Section 3.4, mainly replacing the VDPF with a VIDPF, and using the four consistency checks discussed in Section 3.4.

Private Histogram π_{HIST}

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

- **Input:** Each client C_i has an input point $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$ and $m := |\mathbf{X}|$.
- **Output:** $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$ output a histogram of the ℓ clients' data. If the servers abort then it denotes a malicious server involvement.
- **Primitive:** $\mathsf{VDPF} := (\mathsf{Gen}, \mathsf{BatchEval})$ is a verifiable distributed point function. $\mathrm{H} : \{0,1\}^* \to \{0,1\}^\kappa$ is a random oracle.

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

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

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

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

2: **Server Computation.**

If this is the first client, each server $\mathcal{S}_b$ initializes $\mathsf{HIST}_{(b,b+1)}$ and $\mathsf{HIST}_{(b+1,b)}$ for $b \in \{0,1,2\}$ as follows:

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

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

(a) **VDPF Evaluation:** Each server $\mathcal{S}_b$ computes $\mathrm{Y}_{(b,b+1)}$ and $\mathrm{Y}_{(b,b+2)}$ for $b \in \{0,1,2\}$ as follows: **(Repeated for ℓ clients)**

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

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

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

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

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

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

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

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

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

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

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

$\mathcal{S}_0$ computes $h_0 := \mathrm{H}(\mathrm{Y}_{(0,1)} - \mathrm{Y}_{(0,2)} \parallel \mathrm{Y}_{(0,2)} - \mathrm{Y}_{(2,1)})$ and $\mathcal{S}_1$ computes $h_1 := \mathrm{H}(\mathrm{Y}_{(2,0)} - \mathrm{Y}_{(1,0)} \parallel \mathrm{Y}_{(1,2)} - \mathrm{Y}_{(2,0)})$.

(b) **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 $u_i := \{(\pi_{(0,1)}, \pi_{(0,2)}, \pi_{(2,1)}, \tau_{(0,1)}, \tau_{(0,2)}, \tau_{(2,1)}, h_0)$ values for client $i \in [\ell]\}$. $\mathcal{S}_1$ sets $v_i := \{(\pi_{(1,0)}, \pi_{(2,0)}, \pi_{(1,2)}, 1-\tau_{(1,0)}, 1-\tau_{(2,0)}, 1-\tau_{(1,2)}, h_1)$ 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: $(\mathsf{ver}, \mathsf{L}) := \pi_{\mathsf{check}}(\mathbf{u}, \mathbf{v})$:

$$\mathsf{ver} := 0 \text{ if there exists a client such that} : (\pi_{(0,1)} \neq \pi_{(1,0)}) \vee (\pi_{(0,2)} \neq \pi_{(2,0)}) \vee (\pi_{(2,1)} \neq \pi_{(1,2)}) \vee$$

$$(\tau_{(0,1)} + \tau_{(1,0)} \neq 1) \vee (\tau_{(0,2)} + \tau_{(2,0)} \neq 1) \vee (\tau_{(2,1)} + \tau_{(1,2)} \neq 1) \vee (h_0 \neq h_1)$$

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

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

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

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

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

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.
The servers locally perform the following computation:
The servers aggregate all correct client inputs into the histogram as follows: **(Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)**

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

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

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

3: **Output Phase**.

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

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

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

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

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

Figure 19: Private Histogram Protocol π_{HIST}.