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”. \( T \)-heavy hitters allow computing the \( T \) most popular responses (for a given threshold \( T \) 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 \( T \) 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,
Table 1: Threat model comparisons, client input validation, and server-to-server communication.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Correctness &amp; Privacy Against Malicious Corruption</th>
<th>Client Input Validation</th>
<th>Low Server-to-Server Communication</th>
<th>No. of Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPF [12, 13, 29]</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>2×</td>
</tr>
<tr>
<td>Poplar (IDPF) [10]</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>2</td>
</tr>
<tr>
<td>Bucketization (DP) [2]</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>2</td>
</tr>
<tr>
<td>MPC-based [8]</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>3</td>
</tr>
<tr>
<td>Sorting-based [4, 32]</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>3</td>
</tr>
<tr>
<td>PLASMA (this work)</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>3</td>
</tr>
</tbody>
</table>

† 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 Poplar also introduced a “Verifiable IDPF (VIDPF)” similar to ours, which guarantees the same security properties. However, their constructions, namely Poplar 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 \(O(\ell)\) to \(O(\ell'(\log_2 \frac{\ell}{\ell'}))\) number of hashes where there are \(\ell'\) 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, \(\ell'\) 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 when regard to runtime by a factor of 5 – 10× over WAN for ℓ = 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× and 45×, respectively. For these conditions, we further analyzed the monetary cost of PLASMA, [4], and Poplar and report that PLASMA is more than 2.5× and 4× cheaper respectively.

Applications. We evaluate PLASMA for two applications: a) detecting frequently visited URLs, and b) identifying popular coordinates.
Poplar and Prio [19]. However, their threat model does not address arbitrary noise to the result without the honest server realizing it. Poplar’s threat model is robust against malicious clients but remains due to the blowup in key size, as the client would need to send new DPF keys for each level, resulting in \(O(n)\) 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 \(T > count\), 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 servers 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 \(T = 1\%\). 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 Bohler 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], Duoran [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 (S₀, S₁, S₂) 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
A maliciously corrupts one of the servers and ℓ′ < ℓ clients.

Clients. Malicious clients may try to deviate from the protocol
to disproportionately 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 pos-
sible 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. Mal-
icious servers ensure 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 pa-
rameters by κ and μ, respectively. Let PRG : \{0, 1\}^κ → \{0, 1\}^{2(κ+1)}
be a pseudorandom generator and Convert : \{0, 1\}^κ → \mathcal{G} be a
map converting a κ-bit string to a pseudorandom group element of
additive group \mathcal{G} (where |\mathcal{G}| > ℓ). We use := for assignment, \overset{r}{\rightarrow_D}
for sampling from distribution \mathcal{D}, = for checking equality, and \| for
catenation. For histograms, we define a public set X with m n-bit
strings as X := \{x₁, x₂, ..., xₘ\} where the i-th string is denoted as
xᵢ for i ∈ [m] and the j-th bit in xᵢ ∈ \{0, 1\}ⁿ is denoted as xᵢ,j for
j ∈ [n]. We denote the first L bits of xᵢ as xᵢ,L := \{xᵢ,1, xᵢ,2, ..., xᵢ,L\}
for L ≤ n. Let S₀ denote the b-th server, for b ∈ \{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 Cᵢ for i ∈ [ℓ]. For an n-bit string a we
represent its bit decomposition as a₁, ..., aₙ ∈ \{0, 1\}. In histograms,
each client Cᵢ has an n-bit input string aᵢ ∈ X, for i ∈ [ℓ], while
aᵢ ∈ \{0, 1\}ⁿ in the case of heavy-hitters. We use a₁, ..., aₙ ∈ \{0, 1\}
to denote the bit representation of the client’s input aᵢ.

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 \(f_{α,β}(x) := β\) if x = α, or 0
otherwise. A DPF consists of two algorithms: Gen and Eval. The
Gen algorithm takes as input the function \(f_{α,β}\) and outputs two
keys key₀ and key₁. The Eval algorithm evaluates an input x such
that \(\text{Eval}(0, \text{key}_0, x) + \text{Eval}(1, \text{key}_1, x) = β\) for x = α, and 0 for
x ≠ α. 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 his-
tograms. Each client holds an n-bit string α and the servers S₀ and
S₁ have a small set X := \{x₁, x₂, ..., xₘ\} of m n-bit strings. Each
client secret shares their input α ∈ X using a DPF as \(\text{Eval}_α(\text{key}_γ, γ) :=
\text{DPF.Gen}(ℓ^γ, α, 1, \mathcal{G})\). The client sends key₀ to S₀ and key₁ to S₁.
Upon receiving the client key, each server S₀ evaluates the DPF on
all m strings of X as y₀ := \{\text{DPF.Eval}(b, \text{key}_b, x_i)\}_{x_i \in \mathcal{X}}
and computes a vector of output shares y₀ ∈ \mathbb{F}^m, for some large enough
finite field \mathbb{F} and m = |\mathcal{X}|. The servers repeat this for multiple
clients and aggregate the y₀ vectors in a counter vector Y₀. Finally,
the servers exchange Y₀ and Y₁ to compute the output histogram
as Y := Y₀ + Y₁. 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 Y₀ and Y₁ 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 x 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 S₀ is malicious and introduces additive
errors (e.g., \(\delta ∈ \mathbb{F}^m\)) in Y₀ ′ := Y₀ + \delta. That way, the output Y of the
histogram would be biased by \(δ\) as Y := Y₀ ′ + Y₁ = Y₀ + Y₁ + \delta. The
honest server S₁ 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 \(O(ℓ)\) 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] (Ap-
pendix 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,
Our work provides the first maliciously secure protocol whose server-to-server communication depends logarithmically on the total number of clients. 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 \( S_2 \) 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 \( \alpha \) (i.e., the pairwise evaluation of the DPF keys on point \( \alpha \) 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 \( \alpha \) 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 \( X \). 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 \( S_3 \) is involved in two of the three sessions: specifically, sessions \( S_1 - S_2 \) and \( S_2 - S_0 \). The client generates (key\((0,1)\), key\((1,0)\)) for session \( S_0 - S_1 \), (key\((1,2)\), key\((2,1)\)) for session \( S_1 - S_2 \), and (key\((0,2)\), key\((2,0)\)) for session \( S_2 - S_0 \). \( S_0 \) receives key\((0,1)\) and key\((1,0)\) from the client for sessions \( S_0 - S_1 \) and \( S_2 - S_0 \), respectively. \( S_1 \) receives key\((1,0)\) for session \( S_0 - S_1 \) and key\((1,2)\) for session \( S_2 - S_1 \), while \( S_2 \) receives key\((2,1)\) and key\((2,0)\) for sessions \( S_1 - S_2 \) and \( S_2 - S_0 \), respectively.

In our optimization, instead of running two sessions in each server, we run all three sessions between \( S_0 \) and \( S_1 \) and use \( S_2 \) as the attestator 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 key\((2,1)\) to server \( S_0 \) and key\((2,0)\) to server \( S_1 \) respectively. The key distribution process by the client is illustrated in Fig. 1.

Our optimization allows \( S_0 \) to replicate the computation of \( S_1 \) in session \( S_1 - S_2 \) (because they both have key\((2,1)\)) and \( S_2 \) acts as an attestator by just sending hashes to \( S_1 \) for the same messages that \( S_0 \) should send. These hashes prevent \( S_0 \) from acting maliciously. Similar protocol steps are run by \( S_2 \) to attest the \( S_2 - S_0 \) session and prevent \( S_1 \) (who is replicating \( S_2 \)) from acting maliciously. This optimization, shown in Fig. 2, allows us to batch-verify all three sessions as a single session between \( S_0 \) and \( S_1 \) using hashes.

**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 \( \pi \) upon evaluating a pair of DPF keys on a given input point. The two servers obtain the same proof \( \pi \) if the client generates the DPF keys honestly (i.e., the DPF output is non-zero only at a single point \( \alpha \)). 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 \( X \)) 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 \( \alpha_1, \alpha_2, \) and \( \alpha_3 \) 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 \( Y_{(0,1)}, Y_{(0,2)}, \) and \( Y_{(2,1)} \) be the output of the VDPF evaluation by \( S_0 \) on keys key\((0,1)\), key\((0,2)\), and key\((2,1)\) corresponding to sessions \( S_0 - S_1 \), \( S_0 - S_2 \), and \( S_2 - S_1 \), respectively. Similarly, let us denote \( Y_{(1,0)}, Y_{(2,0)}, \) and \( Y_{(1,2)} \) be the output of the VDPF evaluation by \( S_1 \) on keys key\((1,0)\), key\((2,0)\), and key\((1,2)\) corresponding to sessions \( S_0 - S_1 \), \( S_0 - S_2 \), and \( S_2 - S_1 \), respectively.
This is performed by querying the oracle

\[ T \]  

number of prefix-count queries are independent of the number of 

\[ \Theta(n) \]  

count oracle queries for heavy hitter string. Hence, the algorithm makes at most 

\[ k \]  

all 

\[ (\alpha_1, \alpha_2, \ldots, \alpha_k) = \text{prefix of length } k \]  

the count of strings that contain the 

\[ \gamma \]  

\[ \mathcal{C}_i \]  

present the histogram protocol in Appendix H.

**Heavy-Hitters from \( T \)-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 \( T \) 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 \( T \) threshold prefix-count queries that return 1 if at least some clients’ input strings contain \( p \), otherwise, it returns 0. We define it as:

**Definition 1** \((T\text{-prefix-count query oracle } \Omega_{\alpha_1, \ldots, \alpha_t}(p, T))\), Return 1 (on input prefix \( p \in \{0, 1\}^* \)) if prefix \( p \) appears at least \( T \) times in the clients’ input strings \( \alpha_1, \alpha_2, \ldots, \alpha_t \in \{0, 1\}^* \) where client \( C_i \) has input string \( \alpha_i \) for \( i \in [t] \), otherwise, return 0.

**\( T \)-Heavy hitters.** The \( T \)-heavy hitters algorithm (for threshold \( T \)) is provided with oracle \( \Omega_{\alpha_1, \ldots, \alpha_t}(p, T) \) for computing \( T \)-prefix count for prefix \( p \) over the client input strings \( \alpha_1, \ldots, \alpha_t \). The initial prefix is the empty string \( e \). At each level \( k \), it considers the heavy-hitter prefixes \( p \in \{0, 1\}^k \) of length \( k \) in \( HH^k \), which contains the list of \( \cdot \) -bit strings that appear at least \( T \) times. The algorithm performs a breadth-first search of the prefix tree. It includes \( k+1 \) bit length strings \( p \in \{0, 1\}^k \) if prefix \( p \) occurs at least \( T \) times in the input strings \( (\alpha_1, \ldots, \alpha_t) \), otherwise, it gets pruned along its subtree. This is performed by querying the oracle \( \Omega_{\alpha_1, \ldots, \alpha_t}(p, T) \). The same process is repeated for \( p \in \{0, 1\}^k \). The algorithm repeats this for all \( \cdot \) -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 \( T \)-heavy hitters. Our formal algorithm is presented in Fig. 3.

**Cost Analysis.** There are \( t \) input strings in total. For any string of length \( k \), there are at most \( \frac{k}{l} \) 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 \( \frac{n}{l} \cdot r \) prefix-count oracle queries for \( n \) levels. If we set the threshold to be a constant fraction of all input strings (e.g., \( T = 0.01t \)), then the number of prefix-count queries are independent of the number of input strings (e.g., \( n_{\text{prefix}} = n_{\text{prefix}}(\alpha, T) = 10^m \)).

**Parameters:** Threshold \( T \in \mathbb{N} \) and string length \( n \in \mathbb{N} \).

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

**Outputs:** The set of \( T \)-heavy hitters strings in \( \alpha_1, \alpha_2, \ldots, \alpha_t \).

**Algorithm:**

1. Initialize \( HH^n := \{HH^0, HH^1, \ldots, HH^n\} := \{(\emptyset), \emptyset, \ldots, \emptyset\} \), where \( HH^0 \) contains empty string \( e \) and \( HH^1, \ldots, HH^n \) are empty sets.
2. For each prefix \( p \in HH^k \) of length \( k \) with \( k = 0, 1, 2, \ldots, n-1 \) and \( b \in \{0, 1\} \):
   - If \( \Omega_{\alpha_1, \ldots, \alpha_t}(p, T) = 1 \), then \( HH^{k+1} := HH^{k+1} \cup \{p \} \).
3. Output \( T \)-heavy hitters \( HH^n := \{HH^0, HH^1, \ldots, HH^n\} \).

**Figure 3:** Algorithm for computing \( T \)-heavy hitters.

### 3.4 \( T \)-Prefix Count Queries Oracle from VIDPF

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

**3.4.1 Verifiable Incremental DPF (VIDPF).** A DPF allows a client to succinctly share a vector of size \( 2^m \) 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 \( 2^m \) 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 \( (\pi_1, \ldots, \pi_t) \) 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 \( st_{k-1} \) and proof \( \pi_{k-1} \), 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 \( st_k \), and a new proof \( \pi_k \) corresponding to VIDPF evaluation of the path involving \( x \).

More formally, we map the high-level ideas of VIDPF using the following two algorithms:

1. \( \text{Gen}(1^*, \alpha, (\beta_1, \beta_2, \ldots, \beta_t), (\gamma, \zeta)) \rightarrow (\text{key}_0, \text{key}_1) \) : Given security parameter \( k \), input size \( n \), input string \( \alpha \in \{0, 1\}^n \), and values \( \beta_1, \ldots, \beta_t \), the key generation algorithm outputs two VIDPF keys \( \text{key}_0 \) and \( \text{key}_1 \).
2. \( \text{EvalPref}(b, \text{key}_b, x, st_{k-1}, \pi_{k-1}) \rightarrow (st_k, y_b, \pi_k) \) : Given a VIDPF key \( \text{key}_b \) 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 \( y_b \in G \), and a proof \( \pi_k \in \{0, 1\}^* \).

Correctness of the VIDPF ensures that for all input points \( \alpha \in \{0, 1\}^n \), output values \( \beta_1, \ldots, \beta_t \in G \), VIDPF keys generated as \( (\text{key}_0, \text{key}_1) \) u Gen(\( \alpha, \beta_1, \beta_2, \ldots, \beta_t \), (\( \gamma, \zeta \))) and all values \( x \in \{0, 1\}^k \), where \( k \leq n \), the following holds for all \( k \leq n \):

\[
\pi_0^k = \pi_k^k \quad \text{and} \quad y(x) = \gamma(x, y_0 + y_1) = \begin{cases} \beta_b, & \text{if } x \text{ is a prefix of } \alpha, \\ 0, & \text{otherwise} \end{cases}
\]

where \( (\pi_0^k, y_0, \pi_0^k) := \text{EvalPref}(0, \text{key}_0, x, st_{k-1}^k, \pi_{k-1}^k) \) and \( (st_1^k, y_1, \pi_1^k) := \text{EvalPref}(1, \text{key}_1, x, st_{k-1}^k, \pi_{k-1}^k) \). 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 \( y_0 \) or key \( y_1 \)
We provide a construction of VIDPF in Figs. 14 and 15 (Appendix B) for the VIDPF keys for the three sessions on \( p \) (containing 1 in this case, along the path), and check 4 ensures that values for the root level (i.e., \( p \)) 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 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 \( y^p \) is verified, the secret shares of \( y^p \) are aggregated into counter \( cnt^p \). The servers repeat the above steps for all the clients in parallel to obtain secret shares of \( cnt^p \). The servers invoke the comparison functionality \( F_{cmp} \) (Fig. 7) with the secret shares of \( cnt^p \) and threshold \( T \). \( F_{cmp} \) reconstructs \( cnt \) and outputs 1 if \( cnt \geq T \), otherwise, it outputs 0. This is returned by the servers as the output of the \( T \)-prefix count oracle query to the string \( p \) \( \parallel 0 \). Similar steps are run for \( p \parallel 1 \). The comparison functionality \( F_{cmp} \) 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 \( F_{cmp} \). If \( cnt \) is inconsistent across any pair of servers then \( F_{cmp} \) 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 \( F_{cmp} \) 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 \( \ell \) leaves for \( \ell \) clients. First, the servers check the equality of \( \ell \) 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 \( O(\ell (\log_{\ell} \frac{\ell}{t})) \), for \( t' \) malicious clients, instead of \( O(\ell) \), while when \( t' = 0 \) our communication is down to \( O(1) \). Formal details can be found in Section 5.
4 PRIVATE HEAVY HITTERS

We provide the ideal functionality $F_{HH}$ for heavy-hitters between three servers and $t$ clients in Fig. 4. Adversary $A$ maliciously corrupts any one of the servers and multiple clients. Note that this corruption can easily happen; if $A$ has maliciously corrupted a server, then $A$ can spawn multiple malicious clients. Additionally, if $A$ controls a server, it can instruct $F_{HH}$ to discard an honest client's input. It can also instruct the functionality to abort at a particular level $k + 1$. In this case, $A$ and the honest servers receive the set of all (that have not been discarded by $A$) $k$-bit heavy-hitting prefixes as output, and the functionality instructs the honest servers to abort. We remark that $F_{HH}$ never leaks an honest client's inputs.

**PARAMETERS:** Servers $S_0, S_1, S_2, t$ clients $C_i$ for $i \in [t]$. $S_0, S_1, S_2$ agree on:
- A bound $b$ on the number of client submissions.
- A bound $T$ on the threshold for heavy hitters.

**INPUTS:** Servers $S_0, S_1, S_2$ do not have any input. Clients $C_i$: A point $\alpha_i \in \{0, 1\}^n$ for $i \in [t]$. $\alpha_i$ represents the $i$th bit of $\alpha_i$.

**OUTPUTS:** Initial: $HH^{(0)} := (HH^0, HH^1, \ldots, HH^n) := \{(1), 0, \ldots, 0\}$. For $k \in \{0, \ldots, n - 1\}$ and for each prefix $p \in HH^k$, update $HH^{k+1} := HH^{k+1} \cup \{(p, b) \mid b \in \{0, 1\}, \forall b \in \{0, 1\}, F_{HH}$ outputs the following:
- Three servers $S_0, S_1, S_2$: Set of $T$-heavy hitters $HH^{(k)}$.
- Clients $C_i$: No output for $i \in [t]$.

**CORRUPTION:** Adversary $A$ maliciously corrupts one server and multiple clients together. $A$ can perform the following:
- If $A$ instructs the functionality to discard the $j$th client's input, then $F_{HH}$ discards $\alpha_j$ from the output computation.
- If $A$ instructs the functionality to abort at level $k + 1$ by sending $(\perp, k + 1)$, then $F_{HH}$ returns $HH^k$ to $A$ and the honest servers; additionally, $F_{HH}$ instructs the honest servers to abort by sending $\perp$.

Figure 4: The ideal $F_{HH}$ functionality for $T$-heavy hitters.

Our detailed protocol $\pi_{HH}$ that implements $F_{HH}$ appears in Figs. 5 and 6, while high-level ideas of our protocol can be found in Sections 3.3 and 3.4. Our $\pi_{HH}$ protocol privately computes all the $T$-heavy-hitting strings (and their heavy-hitting prefixes) given the input data of $t$ clients, while protecting the privacy of the individual data points. $\pi_{HH}$ runs on three servers ($S_0, S_1, S_2$) that utilize our verifiable incremental DPF (VIDPF) protocol to privately aggregate the clients' data points. Specifically, $\pi_{HH}$ 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 $\sigma \in \{0, 1\}^n$, and output value $(\beta^1, \ldots, \beta^m) := (1, \ldots, 1)$ along the path to $\sigma$, using independent randomness for each key generation. Employing three pairs of keys essentially allows us to run three separate VIDPF sessions. $S_0$ and $S_1$ each have one key for each of the three sessions, while $S_2$ acts as a consistency checking server and shares one key with each of the other two servers. More specifically, the client generates $(key_{(0,1)}, key_{(0,2)})$ for $S_0$, $(key_{(1,0)}, key_{(1,2)})$ for $S_1$, and $(key_{(2,1)}, key_{(2,0)})$ for $S_2$. The client sends $(key_{(0,1)}, key_{(0,2)}, key_{(1,2)})$ to $S_0$, $(key_{(1,0)}, key_{(1,2)}, key_{(2,0)})$ to $S_1$, and $(key_{(2,1)}, key_{(2,0)})$ to $S_2$ as shown in Fig. 1.

**Server Computation.** Each server initializes a set of sets for heavy-hitters as $HH^{\leq n} := (HH^0, HH^1, \ldots, HH^n) := \{(1), 0, \ldots, 0\}$, where HH$^0$ is a set with the empty string $\epsilon$, $HH^1, \ldots, HH^n$ are empty sets and $HH^k$ corresponds to the $k$th level. The servers start accepting VIDPF keys from the clients. As in our histogram protocol, $S_2$ acts as an attesting server for the sessions involving keys $key_{(2,0)}$ and $key_{(1,2)}$ 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 $p \in HH^k$, the servers initialize to 0 a cnt$^{p[0]}$ (resp. cnt$^{p[1]}$) 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 $\gamma \in \{0, p \parallel 0, p \parallel 1\}$ for every client in all three sessions. For each client, the servers obtain new evaluation states (corresponding to prefix $\gamma$), VIDPF output for prefix string $\gamma$, 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 $y_{\gamma}^{b}$ and a hash $y_{(b_0,b_1)}$, 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 $y_{\gamma}^{b} + y_{\gamma}^{1}$ for each client to verify that $y_{\gamma}^{b} + y_{\gamma}^{1} = 1$ (i.e., the non-zero root value is 1). If $k \neq 1$, then the servers reconstruct $y_{\gamma}^{b} - (y_{\gamma}^{0} + y_{\gamma}^{1})$ 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 $y_{\gamma}^{b} = (y_{\gamma}^{0} + y_{\gamma}^{1})$. Both these checks are reduced to checking the equality of a string (corresponding to each client) held by servers. Let $u$ (resp. $v$) be the list of $t$ (one for each client) strings held by the first (resp. second) server. Then, the servers perform a batch verification of $u$ and $v$ strings by invoking the subprotocol $\pi_{check}(u, v)$ in Fig. 8. If the two lists $u$ and $v$ are equal then $\pi_{check}$ returns $\perp = 1$, else it returns $\perp = 0$ and a list $L$ containing the indices of elements where the lists differ. This is performed for all three sessions. $S_2$ 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 $L = L \cup L'$ and $L''$ 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 $y_{\gamma}^{b}$ is validated for $\gamma \in \{0, p \parallel 0, p \parallel 1\}$, it is aggregated into cnt$^{b} := cnt^{+} + y_{\gamma}^{b}$. This is locally performed by each server (for all three sessions) using the secret shares of $y_{\gamma}^{b}$ 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 $a_i \in X$ for $i \in \{1\}$.
Output: The servers $S_0$ (for $b \in \{0, 1, 2\}$) output the set of $T$-heavy hitters $HH^T \triangleq \{ F_{HH}(T, T, (a_i)_{i\in\{2\}}) \}$.

Primitive: VIDPF $(Gen, EvalPref, EvalNext)$ is a verifiable incremental DPF. $H_0, H_1 : \{0, 1\}^\ast \rightarrow \{0, 1\}^\ast$ are random oracles.

Client C Computation. (Repeated for $\ell$ clients, each of which has their own private input $a$)

1. Client $C$ with input $a$ prepares three pairs DPF keys with independent randomness $u, v, w \sim \{0, 1\}^\ast$, as follows:
   $$(\{u \_\_\_1 \_\_\_1 \_\_\_2\}, \{v \_\_\_1 \_\_\_1 \_\_2\}, \{v \_\_\_1 \_\_\_2\}) \quad \text{for} \quad \{x \_\_\_1 \_\_\_1 \_\_\_2\}, \{x \_\_\_1 \_\_\_1 \_\_2\}, \{x \_\_\_1 \_\_\_2\}$$
2. The client sends $(u \_\_\_1 \_\_\_1 \_\_2, v \_\_\_1 \_\_\_1 \_\_2, \{v \_\_\_1 \_\_\_1 \_\_2\})$ to $S_0$ and $(v \_\_\_1 \_\_\_2, w \_\_\_1 \_\_\_2, \{w \_\_\_1 \_\_\_2\})$ to $S_2$.

Server Computation.
Each server $S_0$ initializes $HH_0 \triangleq (HH_1, HH_2, \ldots, HH_{\ell}) \triangleq \{\{\\}, \emptyset, \ldots, \emptyset\}$. Repeat the following steps for length of $k$ bits, where $k \in \{0, \ldots, n - 1\}$:

1. Initialization. For prefix $p \in HH_0$; servers initialize the aggregation variables for prefixes $y \in \{p \parallel 0, p \parallel 1\}$ as follows:
   $$s_0 \text{ sets } cm_0^{y} := cm_0^{y_{(0,2,1)}} = 0, \quad s_1 \text{ sets } cm_1^{y} := cm_1^{y_{(0,2,0)}} = 0, \quad s_2 \text{ sets } cm_2^{y} := cm_2^{y_{(0,2,0)}} = 0$$

2. VIDPF Evaluation. For prefix $p \in HH_0$, server $S_0$ computes:
   (a) If $(p \parallel 0)$ then $S_0$ sets $s_0^{y_{(0,2,1)}} := s_0^{y_{(0,2,0)}} := s_0^{y_{(1,2,1)}} := 0$.
      If $(p \parallel 1)$ then $S_0$ retrieves $s_0^{y_{(0,2,1)}}$ and $s_0^{y_{(1,2,1)}}$.
   (b) Each server $S_0$ evaluates the VIDPF on the prefixes $y \in \{p \parallel 0, p \parallel 1\}$ as follows and stores them in memory:
      $$S_0 \text{ sets } cm_0^{y} := cm_0^{y_{(0,2,1)}} = 0, \quad S_1 \text{ sets } cm_1^{y} := cm_1^{y_{(0,2,0)}} = 0, \quad S_2 \text{ sets } cm_2^{y} := cm_2^{y_{(0,2,0)}} = 0$$

3. (Repeats for $\ell$ clients)
   (3a) Let $p \in HH_0$ and $(y, \pi)$ be a non-heavy hitter string.
   (3b) Let $H$ be a heavy hitter string.
   (3c) Repeate the following steps for length of $\ell$ bits.

\[\text{Figure 5: Private $T$-Heavy Hitters Protocol } \eta_{HH} \text{ (continues in Fig. 6).}\]

Pruning. The servers prune to pruning and invoke $F_{\text{CAMP}}$ (Fig. 7) on the secret shares of $\text{cnt}^T$ for $y \in \{p \parallel 0, p \parallel 1\}$ for all sessions and threshold $T$. Based on the output of $F_{\text{CAMP}}$ the following occurs:

- $F_{\text{CAMP}}$ returns 1 if $\text{cnt}^T \geq T$ (i.e., $y$ is a heavy-hitter string).
- In this case, the prefix $y$ is added to the list of $k + 1$-bit heavy-hitter set (i.e., $HH^{k+1} \triangleq HH_{k+1}$).
- $F_{\text{CAMP}}$ returns 0 if $\text{cnt}^T < T$ (i.e., $y$ is a non heavy-hitter string). In this case, the prefix $y$ is ignored.

If $F_{\text{CAMP}}$ returns 1, 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 $F_{\text{CAMP}}$) or it provided incorrect client output shares as input (condition 4 in $F_{\text{CAMP}}$).

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, $F_{\text{CAMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [36].
PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries

Proceedings on Privacy Enhancing Technologies 2024(3)

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 \( \pi_{\text{batch}} \) (Fig. 8):

(a) \( S_0 \) sets \( u_i := \{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}, h(\cdot)^{\text{PID}}\} \) values for client \( i \in \{T\} \). \( S_1 \) sets \( v_i := \{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}, h(\cdot)^{\text{PID}}\} \) values for client \( i \in \{T\} \). \( S_0 \) sets \( u := \{u_i\}_{i \in \{T\}} \) and \( S_1 \) sets \( v := \{v_i\}_{i \in \{T\}} \). \( S_0 \) and \( S_1 \) batch-verify all the client inputs by computing the bit ver and list \( L \) (comprising of invalid client inputs) by running \( \pi_{\text{check}} \) with inputs \( u \) and \( v \) respectively: (ver, \( L \)) := \( \pi_{\text{check}}(u, v) \):

\[
\text{ver} := 0 \quad \text{if} \quad \exists \text{a client whose } (R_{\text{batch}}^{(1)} \neq R_{\text{batch}}^{(2)}) \lor (R_{\text{batch}}^{(2)} \neq R_{\text{batch}}^{(2)}) \lor (h(\cdot)^{\text{PID}} \neq h(\cdot)^{\text{PID}}) \lor (h(\cdot)^{\text{PID}} \neq h(\cdot)^{\text{PID}}),
\]

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

(b) \( S_2 \) possesses \( R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)} \) values for each client. \( S_2 \) verifies that \( S_1 \)’s version of \( R_{\text{batch}}^{(2)} \) matches with \( S_2 \)’s version of \( R_{\text{batch}}^{(2)} \). \( S_2 \) also attests that \( S_2 ’s \) version of \( R_{\text{batch}}^{(2)} \) matches with \( S_0 \)’s version of \( R_{\text{batch}}^{(2)} \) by computing (ver1, \( L' \)) := \( \pi_{\text{check}}(\{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}\}, \{x\}_i \subseteq \{1, 2\} \text{ clients of } \{S_0, S_2\}, \{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}\}, \{x\}_i \subseteq \{1, 2\} \text{ clients of } \{S_0, S_2\}) \).

(c) \( S_2 \) verifies that \( S_1 \)’s version of \( R_{\text{batch}}^{(1)} \) matches with \( S_2 \)’s version of \( R_{\text{batch}}^{(1)} \). \( S_2 \) also attests that \( S_2 ’s \) version of \( R_{\text{batch}}^{(1)} \) by computing (ver1, \( L' \)) := \( \pi_{\text{check}}(\{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}\}, \{x\}_i \subseteq \{1, 2\} \text{ clients of } \{S_0, S_2\}, \{R_{\text{batch}}^{(1)}, R_{\text{batch}}^{(2)}\}, \{x\}_i \subseteq \{1, 2\} \text{ clients of } \{S_0, S_2\}) \).

After batch verification, the servers identify the list of bad clients as \( L := L \lor L' \lor L'' \). The servers ignore the inputs of all clients in \( L \).

(4) Aggregation. Aggregate the VIDPF outputs for prefixes \( y \in \{0, 1\}^k \) as follows: (Repeated for all validated clients in \( \{T\} \setminus L \))

\[
S_0 \text{ sets } cnt^y_{(1)} := cnt^y_{(1)} + y^\prime_{(1)}, \quad cnt^y_{(2)} := cnt^y_{(2)} + y^\prime_{(2)}, \quad \text{and } cnt^y_{(2, 1)} := cnt^y_{(2, 1)} + y^\prime_{(2, 1)};
\]

\[
S_1 \text{ sets } cnt^y_{(1)} := cnt^y_{(1)} + y^\prime_{(1)}, \quad cnt^y_{(2)} := cnt^y_{(2)} + y^\prime_{(2)}, \quad \text{and } cnt^y_{(2, 1)} := cnt^y_{(2, 1)} + y^\prime_{(2, 1)};
\]

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

(5) Pruning. For every \( (k + 1) \)-bit string \( y \), the servers invoke \( F_{\text{CMAP}} \) functionality (Fig. 7) with the additive shares of the node frequency.

\[
S_0 \text{ invokes } F_{\text{CMAP}}(cnt^y_{(1)}, 0, cnt^y_{(2)}, cnt^y_{(2, 1)}, cnt^y_{(2, 0)}), \quad S_1 \text{ invokes } F_{\text{CMAP}}(cnt^y_{(1)}, 0, cnt^y_{(2)}, cnt^y_{(2, 1)}, cnt^y_{(2, 0)}), \quad S_2 \text{ invokes } F_{\text{CMAP}}(0, cnt^y_{(1)}, cnt^y_{(2)}, 0, 0, 0).
\]

The servers abort if \( F_{\text{CMAP}} \) outputs \( 1 \). Otherwise, the servers ignore \( y \) since it is non-heavy hitter.

Output Phase. The servers output \( HH^{\leq n} \) as the set of \( T \)-heavier hitter strings.

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

Figure 7: The ideal \( F_{\text{CMAP}} \) functionality for comparison.

Output Phase. At the end, the servers output \( HH^{\leq n} = \{HH^0, HH^1, \ldots, HH^n\} \) as the set of \( T \)-heavier hitter strings. This completes the description of \( \pi_{HH} \) (Figs. 5, 6).

Theorem 1. Assuming VIDPF is a verifiable incremental DPF and \( H_1, H_2 \) are random oracles, \( F_{\text{CMAP}} \) is a secure comparison functionality (Fig. 7), and \( H \) (in \( \pi_{\text{check}} \)) is collision-resistant, then \( \pi_{HH} \) (Figs. 5 and 6) implements \( F_{\text{HH}} \) in the random oracle, \( F_{\text{CMAP}} \)-model against malicious corruption of one server and \( t \leq t' \) 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 \( t' \leq t \) 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 \( p \parallel 0 \in \{0, 1\}^{k+1} \) by constructing the VIDPF on \( (p \parallel 0, \beta^k) \), i.e., \( f(p \parallel 0) = \beta^k \), where \( \beta^k > 1 \), instead of \( (p \parallel 0, 1) \).

This is detected by the honest servers since they perform a local subtree verification by reconstructing the value \( y^0 = (\beta^k y^0 - y^0 y^1) \) and verifying that it equals 0 for all \( k > 0 \). For \( k = 0 \), the servers verify that \( y^1 = 1 \).

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 \( (a_1, 1) \) for session \( S_0 - S_1 \), \( (a_2, 1) \) for session \( S_1 - S_2 \), and \( (a_3, 1) \) for session \( S_2 - S_0 \), where \( a_1, a_2, a_3 \in \{0, 1\}^n \) and might be different. To ensure the input is consistent across sessions, the servers match the difference of the reconstructed output of \( S_0 - S_1 \) and \( S_2 - S_0 \) session, and the difference of the reconstructed output of \( S_2 - S_0 \) and \( S_1 - S_2 \) 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 \( a_1 = a_2 = a_3 \). This is performed by computing \( \beta^k y^0 \) and \( \beta^k y^1 \) 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:

\( S_0 \) is corrupt. In this case, the session between \( S_1 - S_2 \) is honest. \( S_0 \) runs this session with \( S_1 \) since it obtained key \( k_{(2, 1)} \) from the
client. However, $S_2$ behaves as an attester by sending hashes of the messages that $S_1$ is supposed to send. This forces $S_0$ to act honestly in the $S_1 - S_2$, otherwise, it leads to an abort. Another way a malicious $S_0$ can behave badly is by colluding with a malicious client. The client can provide malformed inputs in $S_0 - S_1/S_2 - S_0$ session or inconsistent inputs across the three sessions. In such a case, a malicious $S_0$ could compute an incorrect hash $h^\parallel_0 \leftarrow H_1(r_0^\parallel_0 \leftarrow y_p^\parallel_0 \leftarrow y_p^\parallel_0)$. This allows $S_0$ to introduce an additive error into the frequency messages that $S_1$ is supposed to send. This forces $S_0$ to attempt to introduce an additive error into the frequency messages that $S_1$ is supposed to send.

$S_2$ is corrupt. In this case, the session between $S_0 - S_1$ is honest. If $S_2$ behaves as a malicious attester by sending incorrect hashes for the $S_1 - S_2$ or $S_2 - S_0$ sessions then the honest servers abort. A malicious $S_2$ can also collude with a malicious client, and the latter can provide malformed inputs in the three sessions. If this happens in the $S_0 - S_1$ session then it gets detected due to verifiability of the VIDPF and the local subtree verification, since both $S_0$ and $S_1$ are honest. If the client provides malformed (e.g., double voting) VIDPF keys $key_{(2,0)}$ and $key'_{(2,0)}$ to $S_1$ and $S_0$ for the sessions involving $S_2$, it again gets detected since $S_0$ computes the hashes $h^\parallel_0$ and $h^\parallel_0$ honestly and $S_1$ 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 phases are locally executed by each server. Each batch-verification step requires $\log_2 r + 1$ 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 $FCMP$ for each prefix, which is performed in parallel for all prefixes at the same level. Instantiating $FCMP$ with Rabbit [36] involves $\log_2 |G|$ rounds, where the frequency count is performed over $G$. Summing up, the best case round complexity of PLASMA is $n \cdot (1 + \log_2 |G|)$ and the worst case round complexity is $n \cdot (\log_2 r + 1 + \log_2 |G|)$. For benchmarking, we implement group $G$ using a 64-bit ring to exploit native CPU ring optimizations.

5 BATCHED CONSISTENCY CHECK

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

Figure 8: $\pi_{check}$ for equality verification of $\ell$ strings between two parties and identification of unequal strings.

As summarized in Fig. 8, $\pi_{check}$ requires K + 1 rounds of communication, where $K = \log_2 r$. The total communicated hashes are roughly $4\ell'(\log_2 r + 2)$, where $u$ and $v$ differ on $\ell'$ elements. It can be further optimized to $2\ell'(\log_2 r + 2)$, 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 $\pi_{hh}$ 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 $k = 128$ 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

1Our 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 $2^n = 256$ 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 $T$ 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 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 perform 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).

**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 $T = 1%$.

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 $\ell'$ to be a 0, 0.01, 0.1, and 0.3 fraction of the total clients $\ell$. We observe that running with $\ell' = 0.01\ell$ has slightly faster performance than 0.1$\ell$, while 0.3$\ell$ exhibits slightly worse performance than 0.1$\ell$. Still, these differences are marginal compared to the total runtime, so we opt for reporting the 0 and 0.1$\ell$ to make the figures more clear.

**LAN Server Runtime.** PLASMA outperforms Poplar in terms of server runtime by $2.7\times$ (64 bits) and $5\times$ (256 bits) for $\ell = 10^6$ clients and $T = 1\%$ of the clients. This improvement is largely attributed to our efficient VDPF-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.

\[\frac{\text{Client Runtime (sec.)}}{\text{Number of clients (n)}}\]

\[\frac{\text{Client Communication (KB)}}{\text{Number of clients (n)}}\]

**(a) Client Runtime**

**(b) Client Communication**

**Figure 9: Comparisons of client costs for PLASMA and Poplar (KB is Kilobytes and \(\mu s\) is microseconds).**

**Figure 10: Server runtime over LAN.**

**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, $T = 1\%$, 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 $\ell' = 0\ell$ and 0.1$\ell$ corrupt clients, hence yielding a 35$\times$ 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$\times$ improvement of PLASMA over [4].

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

**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.

**Applications.** We discuss two real applications:

*Popular URLs.* Each URL is represented as a 256-bit string and 10,000 most popular URLs are computed among 1 million client-submitted URLs, assuming \( T = 1\% \). 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× 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 \( T = 1\% \) 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 \( \mathcal{P}_{CMP} \) 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 \( \mathcal{P}_{CMP} \) 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 \((T_1, T_2, \ldots)\) 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 \( T \) 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.
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:

- VDPF.Gen(1^k, f_a, b) → (key_y, key_y). Given the security parameter 1^k and a function f, output keys key_y, key_y.
- VDPF.BatchEval(b, key_y, X) → (Y_b, τ_b) : For b ∈ {0, 1}, batch verifiable evaluation takes a set X = \{x_0, x_1, \ldots, x_m\}, where each x_i ∈ {0, 1}^n. Outputs Y_b := \{y_{b,1}, y_{b,2}, \ldots, y_{b,m}\}.

Correctness ensures that Y_0 + Y_1 = f_a(b, 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 τ_0 and τ_1 are the same if and only if they have been generated from valid keys key_y and key_y of a point function.

## B VERIFIABLE INCREMENTAL DPF

We present the VDPF construction, denoted as π_VDPF in Figs. 14 and 15. Our VDPF 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 \$z^{(1)}_r$ for level r, runs PRG* and outputs \$x_0 \$bit seed \$z^{(1)}_b\$ for level l and value \$y^{(1)}_b\$. This occurs at the intermediate levels and is performed by executing the “else” part of Convert. \$y^{(1)}_b\$ comes from \$\mathbb{G}\$ since it generates the output \$w_0\$ based on intermediate \$\beta_0\$. At the leaves, the “if” part of Convert is executed where only \$y^{(1)}_b\$ is generated. The security of our protocol is summarized in Theorem 2.

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

We define \$k\$-collision resistant PRG as follows:

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

\[
(A_0 \parallel T_0 \parallel B_0 \parallel T'_0) := PRG(s_0),
\]

\[
(A_1 \parallel T_1 \parallel B_1 \parallel T'_1) := PRG(s_1),
\]

and \$B_0 = B_1\$.

where \$A_0, A_1, B_0, B_1 \in \{0, 1\}^k\$ 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 Resistant PRG). 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\}^k\$ 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 $k$.
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 VIFDP follows from the input privacy of the underlying IDPF protocol from Poplar, which in turn relies on the pseudorandomness of PRG. Adding $cs^{(i)}$ in steps 16-17 (Fig. 14) does not affect the input privacy of the client in the random oracle model since $cs^{(i)} = π_0^{(i)} ⊕ π_1^{(i)}$ is an XOR of two random oracle outputs. Each server will know the preimage of either $π_0^{(i)}$ or the preimage of $π_1^{(i)}$ by evaluating the given VIFDP 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 $π_1^{(i)}$ (assuming it obtained the preimage of $π_0^{(i)}$ by evaluating the VIFDP 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 $cs^{(i)}$ for $i ∈ [n]$. 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 VIFDP keys that evaluate to two non-zero values at level $i$. It means the servers obtain $s_b^{(i)}(u), s_b^{(i)}(u), s_b^{(i)}(u), s_b^{(i)}(v)$ and $s_b^{(i)}(v)$ from Step 11 of EvalNext (Fig. 15) by evaluating on $u$ and $v$ such that the following holds:

  \[ s_b^{(i)}(u) ≠ s_b^{(i)}(v) \text{ and } s_b^{(i)}(u) ≠ s_b^{(i)}(v) \]

  \[ cs^{(i)} = π_0^{(i)}(u) ⊕ π_1^{(i)}(u) = π_0^{(i)}(v) ⊕ π_1^{(i)}(v), \]

  where $π_b^{(i)}(u) := H_b(u, s_b^{(i)}(u))$ and $π_b^{(i)}(v) := H_b(u, s_b^{(i)}(v))$ for $b ∈ \{0, 1\}$. However, this is not possible in the random oracle model since it breaks the XOR-collision-resistance property of the random oracle $H_b$. The adversary cannot find such a set of $s_b^{(i)}(u), s_b^{(i)}(u), s_b^{(i)}(v)$ and $s_b^{(i)}(v)$ 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 $H_b$ in step 12 of the EvalNext algorithm. So, we also rely on the collision resistance property of $H_b$ to argue that it suffices to check the equality of the hash values computed using $H_b$ 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 VIFDP 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 $i+1$. That is, $u ≠ v^{i+1}$. This means that $s_b^{(i)}(v) = s_b^{(i)}(v)$ and $s_b^{(i)}(u) ≠ s_b^{(i)}(u)$ 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:

  - $s_b^{(i)}$ is same for both servers: In this case both $s_b^{(i)}(v) = s_b^{(i)}(v)$ and $t_b^{(i)}(v) = t_b^{(i)}(v)$. Here the input of the server to EvalNext is the same except for the value $v$. Hence, the evaluation algorithm of the servers on input $v$ will be identical except in step 10 where server $b$ obtains $s_b^{(i+1)}$ values such that $b_0^{(i+1)} + t_b^{(i+1)} = 0$. So, the output cannot be non-zero in this case.

  - $s_b^{(i)}$ is different for both servers: In this case, $s_b^{(i)}(v) = s_b^{(i)}(v)$ but $t_b^{(i)}(v) ≠ t_b^{(i)}(v)$. For this to happen there exists $s_b^{(i)}(v), t_b^{(i)}(v), s_b^{(i)}(v), t_b^{(i)}(v)$ such that $(s_b^{(i)}(v), t_b^{(i)}(v))$ and $(s_b^{(i)}(v), t_b^{(i)}(v))$ are obtained by computing PRG on $s_b^{(i)}(v)$ and $s_b^{(i)}(v)$ respectively and applying Step 6 of EvalNext based on $t_b^{(i)}(v)$ and $t_b^{(i)}(v)$ values.

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 that the XOR-collision resistance property suffices. This property is implemented by assuming that $(H_1, H_2)$ are random oracles.

**C. Heavy-Hitters Protocol $\pi_{HH}$ Proof.**

In this section, we formally prove Theorem 1. Security of our protocol relies on the correctness of $\pi_{check}$. $\pi_{check}$ 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 $\pi_{check}$ 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 $\pi_{check}$.

Next, we prove the security of our protocol in the real-ideal world paradigm of Canetti [14]. Let $A$ denote the real-world adversary corrupting one of the servers and $L'$ clients maliciously in the real-world execution of the protocol. Let $\text{REAL}_{A, \pi_{check}}$ denote $A$’s view after participating in the real-world execution. Let simulator $\text{Sim}$ be the ideal-world adversary, which given access to the algorithm of $A$ and functionality $F_{HH}$, produces the ideal world adversarial view $\text{IAD}_{\text{Sim}, F_{HH}}$.

We prove that our protocol $\pi_{HH}$ securely implements $F_{HH}$ functionality by providing an ideal world PPT simulator $\text{Sim}$ for all PPT.
Notation: We denote the private n-bit string α and its bit decomposition as α₁, ..., αₙ ∈ {0, 1}ⁿ.

Primitives: PRG : {0, 1}ᵏ → {0, 1}²ᵏ⁺² is a pseudorandom generator.
H₁ : {0, 1}ⁱ × {0, 1}ⁿ → {0, 1}²ₘ and H₂ : {0, 1}ⁱ → {0, 1}²ₙ are random oracles.

Gen(1, α, (β₁, β₂, ..., βₙ), G):  
1. Sample s₀, s₁ ← {0, 1}ⁱ for b ∈ {0, 1}  
2. Let t₀₀ := 0 and t₁₁ := 1  
3. for i := 1 to n do  
4.   s₀ᵢ ⊕ s₁ᵢ ⊕ sᵢ := PRG(sᵢ⁻¹) for b ∈ {0, 1}  
5. if αᵢ = 0 then Diff := L, Same := R  
6. else Diff := R, Same := L  
7. t₀ᵢ := s₀ᵢ @ sᵢ  
8. t₁ᵢ := s₁ᵢ @ sᵢ  
9. tᵢ := t₀ᵢ ⊕ t₁ᵢ ⊕ tᵢ  
10. sᵢ := sᵢ ⊕ tᵢ  
11. tᵢ := tᵢ ⊕ tᵢ⁻¹ ⊕ tᵢ⁻¹  
12. sᵢ := Convert(sᵢ⁻¹) for b ∈ {0, 1}  
13. Wᵢ := (−1)ᵢ·(βᵢ − Wᵢ⁻¹ + Wᵢ⁻¹)  
14. cw := sᵢ ⊕ tᵢ  
15. πᵢ := H₁(sᵢ⁻¹) ∥ sᵢ⁻¹  
16. csᵢ := πᵢ ⊕ sᵢ⁻¹  
17. key := (s₀ᵢ, s₁ᵢ)  
18. return key, for b ∈ {0, 1}  

Convert₂(sₙ):  
1. Let u := |[2|c|].  
2. if u = 2ᵐ for an integer m then:  
3. Return the group element represented by PRG(sₙ) mod u,  
4. where PRG⁻¹ : {0, 1}ᵐ → {0, 1}ᵐ.  
5. else:  
6. Let n = ⌊log₂ u⌋ + κ.  
7. Return the group element represented by PRG(sₙ) mod u,  
8. where PRG⁻¹ : {0, 1}ᵐ → {0, 1}ᵐ.

Figure 14: Protocol πVIDPF for Verifiable Incremental DPF (continuing from Fig. 15).

adversaries A, and show that the real and ideal world view are indistinguishable, i.e., REAL-A,π₁₀,π₀₁ ≈ IDEAL-Sim_A,π₁₀,π₀₁. We use a sequence of hybrids (i.e., HYBₙ - HYB₀) to prove indistinguishability.

Proof: We first consider the case where A corrupts server S₂ along with ℓ⁰ clients. Then, we consider the case where A corrupts either S₀ or S₁ along with ℓ⁺ clients.

S₂ is corrupt. We provide the formal simulator in Fig. 16 and argue indistinguishability as follows.

HYB₀: The real world execution of the protocol.

HYB₁: Same as HYB₀, except Sim aborts if a malicious client i has provided inconsistent uᵢ and qᵢ inputs to S₀ and S₁ and yet passed the batched consistency check πᵢ₀check. The two hybrids are indistinguishable due to the correctness of πᵢ₀check.

HYB₂: Same as HYB₁, 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 αᵢ is k-bits heavy-hitting, 2) αᵢ ⊥ 0 or αᵢ ⊥ 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.

HYB₃: Same as HYB₂, except Sim invokes F₁₁H with the extracted inputs to obtain the HH⁺₀⁻₀ and sets and simulates F₃CMP based on whether a prefix y is in HH⁺₀⁻₀ or not. The two hybrids are indistinguishable against a corrupt server S₂ in the F₃CMP-model.

HYB₄: Same as HYB₃, except Sim simulates the DPF key generation for the honest clients with input α_i (β₁₋₀, βₙ₋₀) = (1, (1, ..., 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 HYB₃ and HYB₄ are in the F₃CMP-model. This is the ideal world execution of the protocol, completing our simulation.

Either S₀ or S₁ is corrupt. Next, we consider the case where either S₀ or S₁ 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 S₁ is corrupt along with ℓ⁺ clients.)

HYB₀: The real world execution of the protocol.

HYB₁: Same as HYB₀, except Sim aborts if a malicious client i provided values (Rᵢᵏ⁺₀⁻₀, Rᵢᵏ⁻₀⁻₀, Rᵢᵏ⁺₀⁻₁, Rᵢᵏ⁻₀⁻₁) to S₂ and values (Rᵢᵏ⁺₀⁻₀, Rᵢᵏ⁻₀⁻₀, Rᵢᵏ⁺₀⁻₁, Rᵢᵏ⁻₀⁻₁) to S₁ such that they are not equal, and yet client i passed
Figure 16: Simulation Algorithm against malicious corruption of server $S_2$ and $t'$ clients.

the batched consistency check $\pi_{\text{check}}$: The two hybrids are indistinguishable due to the correctness of $\pi_{\text{check}}$.

**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 $a_i$ is k-bits heavy-hitting, 2) $a_i \parallel 0$ or $a_i \parallel 1$ 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 $F_{\text{HH}}$ with the extracted inputs to obtain HH$^{\delta,m}$ set and simulates the $F_{\text{CMP}}$ functionality based on whether a prefix $\gamma$ is in HH$^{\delta,m}$ or not. The two hybrids are indistinguishable against a corrupt server $S_0$ in the $F_{\text{CMP}}$-model.

**HYB$_4$:** Same as HYB$_3$, except Sim simulates the key generation for the honest clients with $\gamma = (a, (f_1, \ldots, f_n)) = (1, (1, \ldots, 1))$ 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 randomly distributed to the actual aggregation counters since HYB$_2$ and HYB$_4$ are in the $F_{\text{CMP}}$-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 root trees $R_0$ 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 S_i

Corruption: ℓ′ number of clients and server S_i are maliciously corrupt. The rest ℓ − ℓ′ clients and servers (S_1, S_2) are simulated by simulator Sim. Without loss of generality, we will assume that S_i is corrupt; in the case where S_i is corrupt is symmetric.

Primitive: VDPF := (Gen, EvalPref, EvalNext) is a verifiable incremental DPF. H_1, H_2 : \{0, 1\}^* \rightarrow \{0, 1\}^n are random oracles.

Client C Computation. (Repeated for ℓ clients)

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

(key_{(1)}, key_{(1,0)}):=\text{Gen}(1,1,1,0,1,1,1,1,1,1,1,1,1,1,1)

(key_{(2)}, key_{(2,0)}):=\text{Gen}(1,1,1,0,1,1,1,1,1,1,1,1,1,1,1)

Sim sends (key_{(1)}, key_{(2,0)}) to S_i, (key_{(1,0)}, key_{(2,2)}) to S_1 and (key_{(2,1)}, key_{(2,2)}) to S_2 on behalf of the client.

If the left (resp. right) subtrees are equal. If the left (resp. right) subtrees are equal. Proceeding iteratively the parties then the left (resp. right) subtrees are equal. If the left (resp. right) subtrees are equal. Proceeding iteratively

Server Computation. (Simulator Sim initializes a list L_{inp} = \{\} and L_{out} = \{\}, and simulates S_i and S_j)

For each corrupt client i, the simulator performs the following for input extraction:

1. Sim extracts the corrupt client’s input (\sigma_i, \mu_i, ... , \mu_i) from the pair of DPF keys - key_{i,1} and key_{i,1}, provided by client i.

2. Sim waits for further instructions from the ideal world adversary HH’.

3. Sim stores the extracted input (after necessary truncation) \sigma_i as the set of \simu_{(i,1)}, \simu_{(i,2)}, \simu_{(i,3)}, \simu_{(i,4)}

Server Computation. (Simulator Sim initializes a list L_{inp} = \{\} and L_{out} = \{\}, and simulates S_i and S_j)

After the running above extraction process for all corrupt clients, Sim invokes T_{HH} with the input list L_{inp} to obtain the output set of ℓ’-heavy hitting prefixes as HH’E_\delta. The functionality T_{HH} waits for further instructions from the ideal world adversary Sim.

Repeat the following steps for length of k bits, where k \in [0, ..., n-1]:

1. Initialization. For prefix p \in HH_1^{k}, Sim initialize server S_i’s and S_j’s aggregation variables for prefixes Y \in \{p \parallel 0, p \parallel 1\} as follows:

Simulated S_1 sets \simu_{(i,1)} := \simu_{(i,0)} := \simu_{(i,2)} := 0. Simulated S_2 sets \simu_{(i,2)} := \simu_{(i,2)} := 0.

2. VDPF Evaluation. For prefix p \in HH_1^{k}, Sim initializes S_i and S_j by running the original protocol steps. (Repeated for ℓ clients)

3. Batch-Verification. (a) Sim simulates the interaction between corrupt server S_i and honest server S_j by following the protocol steps to update list L.
(b) Sim simulates the interaction between corrupt server S_i and honest server S_j by following the protocol steps to update list L. (c) For each client i: Sim verifies that S_i’s version of (R_{\delta_{1}}, R_{\delta_{2}}) matches with S_j’s version of (R_{\delta_{1}}, R_{\delta_{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 \pi_{batch} between S_i and S_j then Sim aborts.
(d) Sim aborts if 3 client i \in L) its input is 0-bits heavy-hitting (i.e., \sigma_i \in HH_2), 2) \sigma_i \parallel 0 or \sigma_i \parallel 1 is not valid, i.e., \sigma_i \in L_{inp} and check i.e., i \notin L.

4. Aggregation. Sim simulates this step for prefixes Y \in \{p \parallel 0, p \parallel 1\} as follows: (Repeated for all validated clients in \{\ell \setminus L\}

Simulated S_1 sets \simu_{(i,1)} := \simu_{(i,0)} := \simu_{(i,2)} := 0. Simulated S_2 sets \simu_{(i,2)} := \simu_{(i,2)} := 0.

5. Pruning. For every (k+1)-bit string Y, Sim simulates the pruning step as follows:

- If Y \notin HH_1^{k+1} then Sim invokes the simulator of F_{MAP} with output 1 s.t. F_{MAP} returns 1 as output to the servers, s.t. Y is included in the list of heavy-hitting strings.
- If Y \notin HH_1^{k+1} then Sim invokes the simulator of F_{MAP} with output 1 s.t. F_{MAP} returns 0 as output to the servers, s.t. Y gets pruned.

Output Phase: Sim outputs HH’E_\delta as the set of ℓ’-heavy hitting strings on behalf of simulated S_i and S_j, and instructs T_{HH} to send output to the honest servers S_i and S_j.

Figure 17: Simulation Algorithm against malicious corruption of server 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 N_0 and N_1 where the two trees differ. Then they match them with their initial lists of input sets u and v to identify the indices where they differ and then store those indices in L.

- \pi_{check} requires K + 1 rounds of communication, where K = \lceil \log_2 \ell \rceil. Next, we demonstrate that if \ell’ out of \ell leaves differ, then the total communication is \mathcal{O}(\ell’(\log_2 \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 [K], there can be at most \min(2^k, \ell’) unequal nodes.

The total communicated hashes are as follows:

\begin{align*}
2 + 4 \times (\min(2^k, \ell’)) & = 2 + 4 \times (1 + 2 + 2^2 + 2^3 + 2^4 + \ldots + 2^{\ell’}) \\
& \leq 2 + 4 \times (2\ell’ + \ell’) \times (1/\log_2 2) \\
& \approx 8\ell’ + 4\ell’ (\log_2 \ell’ + 2). \\
\end{align*}

We observe that the current version of \pi_{check} communicates roughly 4\ell’(\log_2 2 + 2) hashes. This can be further optimized to 2\ell’(\log_2 2 + 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 \ell_i based on some pre-agreed strings \chi_i \in \chi by the servers. This can be beneficial for traffic avoidance since different roads may have different traffic densities. For example, highways are busier than...
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: \( mn\left(\frac{7}{3} + \frac{32}{9} |R| \right) + 3m|\hat{R}| + 2m|\hat{R}'| \) 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 \( 2^x \) size for \( 2^{-x} \) failure probability. This leads us to the optimized malicious sorting protocol communication cost of: \( 2mn\left(\frac{7}{3} + \frac{32}{9} k \right) + 3mk \).

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 \( 4mn \log n \) 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 \( mn \) 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., \( 2X \) 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:

\[ 2mn\left(\frac{7}{3} + \frac{32}{9} k \right) + 3mk + 8mn \log n + 6mn \text{ bits.} \]

Setting the security parameter \( k \) 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:

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

Therefore, the total server-server communication cost is at least 14.96 \( \times 3 \approx 45 \) gigabytes for computing the heavy hitters over 256-bit keys between three servers for 10^6 clients.
Private Histogram $\pi_{HIST}$

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

- **Input**: Each client $C_i$ has an input point $x_i \in X$ for $i \in [\ell]$ and $m \Rightarrow [X]$.
- **Output**: $S_0, S_1$ output a histogram of the $\ell$ clients’ data. If the servers abort then it denotes a malicious server involvement.
- **Primitive**: VDPF := $\{\text{Gen, BatchEval}\}$ is a verifiable distributed point function. $H : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a random oracle.

1. **Client $C$ Computation.**

   (Repeated for $\ell$ clients, each of which has their own private input $x_i$.)
   
   (a) Client $C$ with input $x_i$ prepares three pairs of DPF keys with independent randomness $u, v, w \leftarrow \{0, 1\}^*$, as follows:
      
      \[(\text{key}(x_i), \text{key}(y_i)) \Rightarrow \text{Gen}(1^*; x_i, u, v) \quad \text{and} \quad (\text{key}(x_i), \text{key}(y_i)) \Rightarrow \text{Gen}(1^*; x_i, u, v)\]
      
   (b) The client sends $(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i))$ to $S_0$, $(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i))$ to $S_1$, and $(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i))$ to $S_2$.

2. **Server Computation.**

   If this is the first client, each server $S_0$ initializes $HIST_{(b, b+1)}$ and $HIST_{(b+1, b)}$ for $b \in \{0, 1, 2\}$ as follows:
   
   - $S_0$ initializes $HIST_{(0, 1)} := \emptyset^m$, $HIST_{(0, 2)} := \emptyset^m$, and $HIST_{(1, 1)} := \emptyset^m$.
   - $S_0$ initializes $HIST_{(2, 1)} := \emptyset^m$, $HIST_{(2, 2)} := \emptyset^m$, $S_1$ initializes $HIST_{(2, 0)} := \emptyset^m$ and $HIST_{(2, 1)} := \emptyset^m$.

   (a) **VDPF Evaluation:** Each server $S_0$ computes $Y_{(b, b+1)}$ and $Y_{(b+1, b)}$ for $b \in \{0, 1, 2\}$ as follows:
      
      Each server $S_b$ parses $r_{(b, b+1)}$ and $r_{(b+1, b)}$ for $b \in \{0, 1, 2\}$ as follows:
      
      $S_0$ parses $Y_{(0, 1)} := \{y_{(0, 1)}, y_{(0, 2)}, \ldots, y_{(1, m)}\}$ and computes $t_{(0, 1)} := \sum y_{(0, 1)}/y_{(1, j)}$.
      
      $S_1$ parses $Y_{(1, 1)} := \{y_{(1, 1)}, y_{(1, 2)}, \ldots, y_{(2, m)}\}$ and computes $t_{(1, 1)} := \sum y_{(1, 1)}/y_{(2, j)}$.
      
      $S_2$ parses $Y_{(2, 1)} := \{y_{(2, 1)}, y_{(2, 2)}, \ldots, y_{(3, m)}\}$ and computes $t_{(2, 1)} := \sum y_{(2, 1)}/y_{(3, j)}$.
      
   (b) **Batch-Verification:** The servers batch-verify the client inputs for all three sessions and across the three sessions by invoking $\pi_{check}$ (Fig. 8):
      
      (i) $S_0$ sets $u := \{(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i)); \text{key}(y_i)\}$ values for client $i \in [\ell]$. $S_1$ sets $u := \{(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i)); \text{key}(y_i)\}$ values for client $i \in [\ell]$. $S_0$ sets $v := \{(\text{key}(x_i), \text{key}(y_i), \text{key}(z_i)); \text{key}(y_i)\}$ and $S_1$ batch-verify all the client inputs by computing the bit test and list $L$ (comprising of invalid client inputs) by running $\pi_{check}$ with inputs $u$ and $v$ respectively: (ver, $L$) := $\pi_{check}(u, v)$.
      
      (ii) $S_0$ possesses $\pi_{(0, 2)}$, $\pi_{(1, 2)}$, $\pi_{(2, 0)}$, $\pi_{(2, 1)}$, $\pi_{(2, 2)}$ values for each client. $S_0$ verifies that $S_0$’s version of $\pi_{(2, 2)}$, $\pi_{(2, 1)}$ matches $S_0$’s version of $\pi_{(2, 1)}$, $\pi_{(2, 2)}$. $S_2$ also attests that $S_2$’s version of $\pi_{(0, 2)}$, $\pi_{(0, 1)}$, $\pi_{(0, 0)}$, $\pi_{(1, 0)}$, $\pi_{(1, 1)}$, $\pi_{(2, 0)}$, $\pi_{(2, 1)}$, $\pi_{(2, 2)}$ by computing $\pi_{(ver, L)}$ as follows:
      
      (iii) $S_2$ verifies that $S_2$’s version of $\pi_{(2, 2)}$, $\pi_{(2, 1)}$, $\pi_{(1, 2)}$, $\pi_{(1, 1)}$, $\pi_{(0, 2)}$ matches $S_2$’s version of $\pi_{(2, 2)}$, $\pi_{(2, 1)}$. $S_2$ also attests that $S_2$’s version of $\pi_{(0, 1)}$, $\pi_{(0, 2)}$, $\pi_{(0, 0)}$, $\pi_{(1, 0)}$, $\pi_{(1, 1)}$, $\pi_{(2, 0)}$, $\pi_{(2, 1)}$, $\pi_{(2, 2)}$ by computing $\pi_{(ver, L)}$ as follows:

      After batch verification, the servers identify the list of bad clients as $L := L \cup L'$.

3. **Output Phase.**

   (a) Each two servers $S_0$ and $S_1$ exchange $H(HIST_{(b, b+1)}; r_{(b, b+1)})$ and $H(HIST_{(b+1, b); r_{(b+1, b)})}$ for random $r_{(b, b+1)}, r_{(b+1, b)} \leftarrow \{0, 1\}^*$.
   
   (b) $S_0$ sends $HIST_{(0, 1)}, HIST_{(0, 2)}, HIST_{(1, 2)}, HIST_{(1, 0)}, HIST_{(2, 1)}, HIST_{(2, 2)}, HIST_{(2, 0)}$, $r_{(2, 1)}$, $r_{(2, 0)}$ to $S_0$. $S_1$ sends $HIST_{(1, 1)}, HIST_{(1, 2)}, HIST_{(2, 1)}, HIST_{(2, 2)}, HIST_{(2, 0)}$, $r_{(2, 1)}$, $r_{(2, 0)}$ to $S_1$. $S_2$ broadcasts $(r_{(2, 0)}, r_{(2, 0)}).
   
   (c) $S_0$ and $S_1$ verify the above hashes. If any of the hashes fail then the servers abort. Else, they perform the following:
      
      (d) $S_0$ and $S_1$ abort if $HIST_0 \neq HIST_1$ or $HIST_1 \neq HIST_2$. Else, they output $HIST$ where $HIST = HIST_0 = HIST_1 = HIST_2$.

Figure 19: Private Histogram Protocol $\pi_{HIST}$. 

24