Vogue: Faster Computation of Private Heavy Hitters

Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal and Somya Sangal

Abstract—Consider the problem of securely identifying τ-heavy hitters, where given a set of client inputs, the goal is to identify those inputs which are held by at least τ clients in a privacy-preserving manner. Towards this, we design a novel system Vogue, whose key highlight in comparison to prior works, is that it ensures complete privacy and does not leak any information other than the heavy hitters. In doing so, Vogue aims to achieve as efficient a solution as possible. To showcase these efficiency improvements, we benchmark our solution and observe that it requires around 14 minutes to compute the heavy hitters for τ = 900 on 256-bit inputs when considering 400K clients. This is in contrast to the state of the art solution that requires over an hour for the same. In addition to the static input setting described above, Vogue also accounts for streaming inputs and provides a protocol that outperforms the state-of-the-art therein. The efficiency improvements witnessed while computing heavy hitters in both, the static and streaming input settings, are attributed to our new secure stable compaction protocol, whose round complexity is independent of the size of the input array to be compacted.

Index Terms—private heavy hitters, secure stable compaction, secure multiparty computation

1 Introduction

MANY real world applications require performing analysis on data aggregated across users (clients). This data aggregation, followed by the analysis of the same, allows deriving useful statistics regarding the clients. For example, consider the scenario where a web browser vendor is interested in finding the most widely used homepages across its users. Obtaining such statistics facilitates the vendor in providing a personalized user experience. The data collection is depicted in Fig. 1, where the web browser vendor acts as the data aggregator to obtain URLs as inputs from clients to determine those URLs whose frequency exceeds a predetermined threshold. Such popularly occurring client inputs (URLs in this case) are referred to as *heavy hitters*. Identification of such heavy hitters also finds use in several other domains–(i) web browser vendors are interested in determining popular web pages that lead to crashes, (ii) to identify potential threats, network service providers require a mechanism to detect popular entities with unusually high traffic [2], (iii) optimizing search engines involves identifying frequent user queries, popular links and advertisements [3], etc. In the above instances, popular web pages, high-traffic entities, and popular queries and links constitute the heavy hitters. Identifying these heavy hitters requires the data aggregator to access each of the client's inputs on clear. Since the inputs comprise private information (e.g., client's choice of homepage), revealing this on clear to the data aggregator compromises client privacy. Hence, it is imperative to design privacy-preserving techniques to identify heavy hitters such that the data aggregator identifies the heavy hitters while ensuring no other information, including the clients' inputs, is leaked.

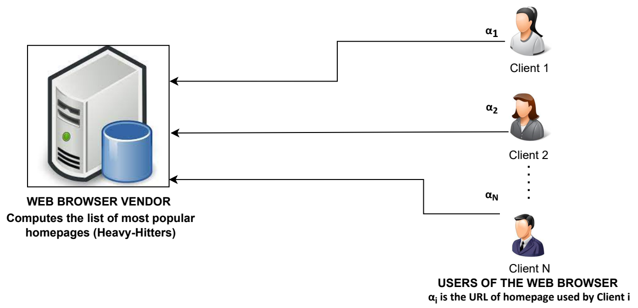
α_1
Client 1
α_2
Client 2
α_N
Client N
WEB BROWSER VENDOR
Computes the list of most popular homepages (Heavy-Hitters)
USERS OF THE WEB BROWSER
α_i is the URL of homepage used by Client i

Fig. 1: Example for heavy hitter computation

Towards addressing this problem, we design a novel system, Vogue, for securely determining heavy hitters. Informally, given a set of $\mathcal{N}$ clients, each of which holds an ℓ-bit input string, a *τ-heavy hitter* is defined as that string which is held by at least τ clients. To securely identify τ-heavy hitters, we rely on the cryptographic technique of secure multiparty computation (MPC). This technique enables a set of n parties to jointly compute a function on their private inputs while guaranteeing that no subset of at most $t < n$ parties controlled by an adversary learn anything other than the function output. Thus, departing from the centralized approach of having a single data aggregator, we move to the distributed setting, where a set of hired servers collectively effectuate the role of the aggregator. This translates to each client secret sharing its input to the set of computing servers, such that even if $\leq t$ servers collude, their shares

This is the extended version of the results that appeared in the poster [1]

- *Pranav Jangir is with New York University. However, this work was carried out during his affiliation at Indian Institute of Science Bangalore, India, as a research associate.*
  E-mail: jangir.pranav@gmail.com
- *Nishat Koti, Varsha Bhat Kukkala, Arpita Patra and Bhavish Raj Gopal are with Indian Institute of Science Bangalore, India.*
  E-mail: {kotis, varshak, arpita, bhavishraj}@iisc.ac.in
- *This work was carried out during Somya Sangal's affiliation at Indian Institute of Science Bangalore, India, as a research associate. E-mail: somyasangal1996@gmail.com*

do not leak any information regarding the client input. The servers carry out the MPC protocol on the input shares to securely compute shares of the heavy hitters. The servers then reconstruct the output towards the data aggregator. The use of MPC guarantees the correctness of the computed output as well as privacy, i.e., the computing servers and the web browser vendor do not learn the client inputs nor any intermediate information. A schematic representation of this is depicted in Fig. 2.

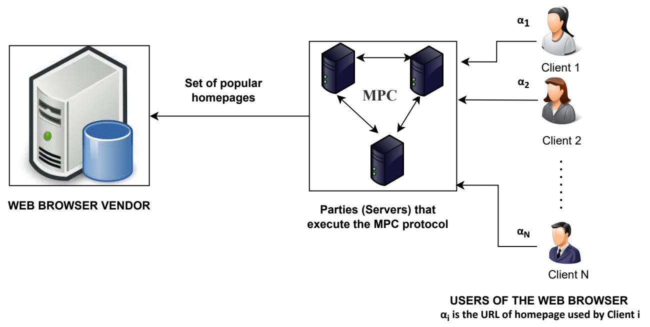

Fig. 2: Example of private heavy hitter computation

We not only focus on designing a secure system, we also aim to design as efficient a solution as possible. Specifically, we aim to achieve the following desirable properties:

- Security against malicious servers: The protocol must be secure even when a malicious adversary controlling a subset of servers arbitrarily deviates from protocol specification. Failing to do so is detrimental for overall system's reliability.
- Security against malicious clients: In addition to malicious servers, it is desirable for the system to guarantee robustness against malicious clients, who may try to send ill-formed inputs to influence the system and tamper with the output. Further, the system must guarantee security even when a subset of clients collude with corrupt servers.
- Client-side efficiency: Another essential factor to consider is the client's computation and communication cost. It is desirable to minimize this overhead at the client to facilitate the participation of low-end devices as well.
- Server-side efficiency: This directly impacts the response time of the system, which is defined as the total time taken from submission of the client's input to the servers, processing of the input, to delivery of the output. A quick response time allows for expediting the necessary action to be taken after obtaining the output. Hence, it is essential to design a system that has efficient server-side computation.

Before we detail our solution that achieves the above properties and contributions therein, we describe the relevant literature that considers private heavy hitters.

1.1 Related Work

The work of [4] provides a system called STAR to compute τ heavy hitters and operates in the single server setting. Here, the client's input is hidden from the servers until the input string is identified as a τ heavy hitter. Hence, this work comes close to the centralized model, albeit offering better privacy. However, as described in [4], the protocol suffers from information leakage, where the server learns the set of clients that have the same input. In fact, the protocol is designed to leverage such leakage of information to achieve improved efficiency. In this way, the protocol trades off privacy for efficiency, as opposed to the current work where the servers learn no information apart from the desired output. Departing from the centralized single server setting, the work of [5] provides a system called Poplar, which operates in the distributed setting with two malicious data collecting servers. Thus, Poplar comes closer to the current work. However, similar to STAR, Poplar also suffers from information leakage, where it leaks the distribution of the clients' strings to the servers. Specifically, it leaks all the τ heavy hitter prefixes and their count. The authors claim that the leakage is modest and capture the same via a leakage function. Further, to protect against this leakage, Poplar suggests the use of differential privacy.

There are also works in the literature that consider the specific case of identifying heavy hitters for *streaming* inputs [6], [7], [8]. Here, the client inputs are assumed to arrive at discrete time steps and the goal is to identify the top K client strings which are popular in the entire stream of inputs seen so far. Since the output is influenced each time new clients' inputs arrive, it is required to re-identify the heavy hitters once sufficiently many new clients have arrived. This is unlike the setting described previously, which we refer to as *static* setting, where all the client inputs are available prior to the computation, and hence the output is computed only once. MPC based solutions to address the streaming setting have also been previously considered [9], [10], [11]. These provide differentially private output since the streaming setting demands additional privacy guarantees as justified in §4. Among these, the work of [11] forms state-of-the-art and improves the efficiency in comparison to prior works.

1.2 Our Contributions

With the previously stated desirable properties in mind, we design Vogue, a system to compute private heavy hitters (PHH). It operates over the ring algebraic structure in the threshold-optimal setting of 3-parties assuming an honest majority (i.e., $t < n/2$) and provides malicious security. Our detailed contributions are listed below.

Private heavy hitters: We propose a novel algorithm for computing heavy hitters in the *static* setting where the set of $\mathcal{N}$ client inputs is available at hand. Our algorithm is highly intuitive and relies on the primitives depicted in Fig. 3. A naive implementation of the protocol using secure versions of these primitives found in the literature [12], [13] is highly inefficient. To ameliorate the situation, we design an improved protocol for secure compaction, details of which are described later. Additionally, since our protocol makes black-box calls to the underlying primitives, any enhancement in efficiency of these primitives is reflected duly in the performance of our protocol. Further, the modular design of our protocol and its generality allows instantiating it with any MPC. Thus, it can inherit security guarantees of the underlying MPC. In this way, instantiating our protocol with the appropriate MPC allows achieving the strongest security notion of guaranteed output delivery (GOD), which ensures delivery of output regardless of any adversarial misbehavior.

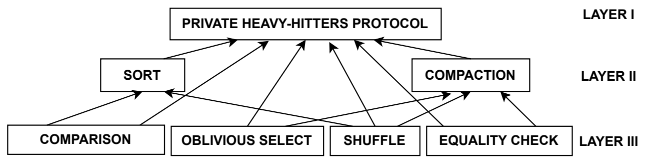

Fig. 3: Building blocks of PHH protocol

Although the problem of computing private heavy hitters was considered in the work of Poplar [5], our protocol improves over it in the following ways.

- Security against malicious servers and clients: Unlike Poplar that only guarantees privacy in the presence of a malicious adversary, Vogue guarantees privacy as well as correctness of output. This holds true even when a subset of malicious clients collude with the malicious server.
- Client-side efficiency: Vogue focuses on minimizing the client-side computation time as well as the client-to-server communication. Specifically, a client is required to communicate $\mathcal{O}(\ell)$ bits in comparison to Poplar, which requires $\mathcal{O}(\ell + \log \mathcal{N})$ bits to be communicated from a client to server. Here, $\mathcal{N}$ denotes the number of clients, and ℓ denotes the bit length of the client's input string. With respect to the computation, we note that Vogue entails the client performing simple operations such as XORs, unlike Poplar, which requires expensive operations such as generation of distributed point function (DPF) keys. This is corroborated by our experimental results where the computation overhead at client is 5.82 ($\times 10^{-6}$) seconds in Vogue, whereas it is 173.1 ($\times 10^{-6}$) seconds in Poplar for an input of 256 bits.
- Server-side efficiency: When designing Vogue, several parameters are taken into account to ensure server-side efficiency. We not only focus on improving the round complexity but also ensure to minimize the computation complexity, both of which play a key role in ensuring a fast response time. In a typical setup where the number of clients $\mathcal{N} = 400,000$ and the client's input size is $\ell = 256$ bits, determining the heavy hitter strings which are held by at least 900 clients, results in a response time of around 14 minutes in Vogue when instantiated with the MPC of [14]. On the contrary, Poplar requires around 86 minutes for the same. Thus, we observe a gain of up to $6\times$ over Poplar.

The comparison of the round and communication complexity of Vogue with Poplar is provided in Table 1.

Parameter	Vogue	Poplar [5]
Client-to-server communication	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell + \mathcal{N})$
Server-to-server communication	$\mathcal{O}(\mathcal{N} \log(\mathcal{N})\ell)$	$\mathcal{O}(\mathcal{N}\ell)$
Server-side round complexity	$\mathcal{O}(\log(\mathcal{N}))$	$\mathcal{O}(\ell)$

TABLE 1: Comparison of Vogue with Poplar.

Finally, unlike [5], our protocol does not leak any information (such as τ heavy prefixes or count of clients possessing the same input) other than the τ heavy hitters, which is achieved for the first time. In fact, our protocol allows keeping threshold τ hidden from computing servers. This is necessary for scenarios where external servers are hired to carry out the computation (e.g., secure outsourced computation), and τ should be hidden from them.

Private heavy hitters - The streaming case: We also consider the case where inputs arrive in a streaming fashion and design a secure protocol to identify the top K strings in the input stream seen so far. In this setting, the work of [11] provides a $\mathcal{O}(K)$ round protocol. Our contribution lies in designing an improved protocol whose round complexity is independent of K. Due to lack of public code for the protocol in [11], we benchmark their solution in our 3PC setting [14] to draw a fair comparison. The results showcase that our protocol has an improvement of up to $3.5\times$ in response time.

Secure stable compaction: Our secure protocol for private heavy hitters, both in the static as well as the streaming setting, requires a secure protocol for *stable* compaction. Elaborately, given a vector $\mathbf{t}$ of 0s and 1s, compaction is defined as the process of reordering the elements in $\mathbf{t}$ such that all 1s appear before the 0s. Further, compaction is said to be stable if the relative ordering among all the 1s (and 0s) is preserved. Several works in the literature have considered designing secure compaction protocols [15], [16], [17], [18]. However, they are either not stable or inefficient. The most recent work of [12] gives a stable protocol with linear communication and round complexity in the number of elements in $\mathbf{t}$. We take a step ahead and design a secure, stable compaction protocol that has round complexity independent of the size of $\mathbf{t}$. Further, our compaction protocol makes no additional assumptions regarding the distribution of 1s and 0s being available publicly, unlike in [12]. In this way, our protocol protects sensitive information that can be derived from the input distribution, thereby providing better privacy guarantees than that of [12]. It is worthwhile to note here that our protocol is generic and modular, which allows generalizing it to an arbitrary number of parties. Complexity comparison of our protocol with [12] is provided in Table 2.

Parameter	Ours	[12]
Round complexity	$\mathcal{O}(\log(\ell))$	$\mathcal{O}(\mathcal{N} \log(\mathcal{N}))$
Communication complexity	$\mathcal{O}(\mathcal{N}\ell)$	$\mathcal{O}(\mathcal{N}\ell)$
Leaks #1's ?	×	✓

TABLE 2: Comparison of our compaction protocol with [12].

1.3 Organization

The paper is organized as follows. We begin with the prelims in §2. This is followed by our secure protocol for computing heavy hitters in the static setting, as well as the secure compaction protocol in §3. Following this, the secure protocol for the streaming case is described in §4. To establish the performance improvement of our solution, we benchmark it and compare it with the state-of-the-art solutions in §5.

2 Preliminaries

Threat model. While our protocols are generic and can be instantiated with any MPC framework which provides support for the primitives described in Table 3, we restrict our focus to the threshold optimal honest majority setting of 3-party computation (3PC) for efficiency reasons. We instantiate the 3PC using the robust framework of [14]. Let $\mathcal{P} = \{P_0, P_1, P_2\}$ denote the set of three parties from which at most one can be maliciously corrupted by a static, probabilistic, polynomial-time adversary $\mathcal{A}$. We assume that parties are connected via pairwise private and authentic channels.

Secret sharing scheme. The following different types of sharing semantics are used in [14].

∘ $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be (3, 1) replicated secret shared or $[\cdot]$-shared, if there exists $[\mathsf{v}]_{01}, [\mathsf{v}]_{02}, [\mathsf{v}]_{12} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_{01} + [\mathsf{v}]_{02} + [\mathsf{v}]_{12}$, and each $[\mathsf{v}]_{ij} \in \{[\mathsf{v}]_{01}, [\mathsf{v}]_{02}, [\mathsf{v}]_{12}\}$ is held by $P_i, P_j \in \mathcal{P}$.

∘ $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if there exists $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared, and there exists $\beta_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ which is held by all parties in $\mathcal{P}$. We let $[\![\mathsf{v}]\!]_i$ denote the shares held by $P_i \in \mathcal{P}$.

The above sharing over the arithmetic ring $\mathbb{Z}_{2^\ell}$ is defined as arithmetic sharing. Analogously, sharing over the Boolean ring $\mathbb{Z}_2$, where addition operations are replaced by XOR, is defined as Boolean sharing ($[\cdot]^{\mathbf{B}}$ and $[\![\cdot]\!]^{\mathbf{B}}$). An ℓ-bit value v is said to be $[\![\cdot]\!]^{\mathbf{B}}$-shared (or equivalently $[\cdot]^{\mathbf{B}}$-shared) if each bit in v is $[\![\cdot]\!]^{\mathbf{B}}$-shared ($[\cdot]^{\mathbf{B}}$-shared). Henceforth, we use shares and secret-shares interchangeably.

Shared key setup. Parties use a one-time key setup [14], [19], [20], [21], [22] to establish common random keys for a pseudo-random function (PRF) between them. This is modelled as a functionality $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 4). This enables each subset of parties to non-interactively sample a common random ℓ-bit string $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \rightarrow X$ be a pseudo-random function (PRF), with $X = \mathbb{Z}_{2^\ell}$. To enable parties to sample common random values non-interactively, the following keys (for the PRF F) are established between the parties: each pair of parties $P_i, P_j \in \mathcal{P}$ know a common k_{ij}, and all parties in $\mathcal{P}$ know $k_{\mathcal{P}}$. P_i, P_j can now sample a common value $r \in \mathbb{Z}_{2^\ell}$, non-interactively, by computing $F_{k_{ij}}(id_{ij})$. Here, id_{ij} denotes a counter maintained by P_i, P_j, which is updated after every PRF invocation.

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

$\mathcal{F}_{\mathsf{Setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Setup}}$ picks random keys k_{ij} for $i, j \in \{0, 1, 2\}$, $i < j$, and $k_{\mathcal{P}}$. Let y_x denote the keys corresponding to party P_x. Then
– $\mathsf{y}_x = (k_{01}, k_{02}$ and $k_{\mathcal{P}})$ when $P_x = P_0$.
– $\mathsf{y}_x = (k_{01}, k_{12}$ and $k_{\mathcal{P}})$ when $P_x = P_1$.
– $\mathsf{y}_x = (k_{02}, k_{12}$ and $k_{\mathcal{P}})$ when $P_x = P_2$.
Output: Send $(\mathsf{Output}, \mathsf{y}_x)$ to every $P_x \in \mathcal{P}$.

Fig. 4: Ideal functionality for shared-key setup

Primitives. In addition to the primitives in Table 3, Vogue also relies on a secure protocol for shuffle and sort. These are modeled via the ideal functionalities $\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Sort}}$ as described in Fig. 5, Fig. 6, respectively. In our work, we instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ using the shuffle protocol from [13]. To obtain a secure protocol for $\mathcal{F}_{\mathsf{Sort}}$, we follow the shuffle-then-sort paradigm. Here, secure sort is realized by first shuffling the input followed by performing an insecure sort, i.e., the sorting protocol need not be data oblivious. For this, we instantiate shuffle protocol via that of [13] followed by performing quicksort[1].

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

Let $\mathcal{S}$ denote the ideal world adversary. $\mathcal{F}_{\mathsf{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of the input vector $\mathbf{a}$ from all parties.
$\mathcal{F}_{\mathsf{Shuffle}}$ proceeds as follows.
– Reconstruct input $\mathbf{a}$ using $[\![\cdot]\!]$-shares of the honest parties.
– Sample a random permutation π from the space of all permutations and generate $\mathbf{a}_o = \pi(\mathbf{a})$.
– Generate $[\![\cdot]\!]$-shares of $\mathbf{a}_o$ and send $(\mathsf{Output}, [\![\mathbf{a}_o]\!]_s)$ to $P_s \in \mathcal{P}$.

Fig. 5: Ideal functionality for shuffle

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

Let $\mathcal{S}$ denote the ideal world adversary. $\mathcal{F}_{\mathsf{Sort}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of the input vector $\mathbf{a}$ from all parties.
$\mathcal{F}_{\mathsf{Sort}}$ proceeds as follows.
– Reconstruct input $\mathbf{a}$ using $[\![\cdot]\!]$-shares of the honest parties.
– Sort $\mathbf{a}$ to generate $\mathbf{a}_o$.
– Generate $[\![\cdot]\!]$-shares of $\mathbf{a}_o$ and send $(\mathsf{Output}, [\![\mathbf{a}_o]\!]_s)$ to $P_s \in \mathcal{P}$.

Fig. 6: Ideal functionality for sort

Protocol	Functionality output
$\Pi_{\mathsf{Select}}([\![a]\!], [\![b]\!], [\![c]\!]^{\mathbf{B}})$	$[\![a]\!]$ if $c = 1$, else $[\![b]\!]$
$\Pi_{\mathsf{Cmp}}([\![a]\!], [\![b]\!])$	$[\![1]\!]^{\mathbf{B}}$ if $a < b$ else $[\![0]\!]^{\mathbf{B}}$
$\Pi_{\mathsf{Eq}}([\![a]\!], [\![b]\!])$	$[\![1]\!]^{\mathbf{B}}$ if $a = b$ else $[\![0]\!]^{\mathbf{B}}$
$\Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	$[\![\mathsf{b}]\!]$; arithmetic sharing of bit $\mathsf{b} \in \mathbb{Z}_2$
$\Pi_{\mathsf{mult}}([\![a]\!], [\![b]\!])$	$[\![c]\!]$ where $c = a \cdot b$
$\Pi_{\mathsf{dotp}}([\![\mathbf{a}]\!], [\![\mathbf{b}]\!])$	$[\![c]\!]$ where $c = \sum_{i=1}^{\mathcal{N}} a_i \cdot b_i$

TABLE 3: Building blocks used in Vogue. Here, $a, b, c \in \mathbb{Z}_{2^\ell}$ and $\mathbf{a}, \mathbf{b}$ denote $\mathcal{N}$-sized vectors.

Differential privacy. When operating on sensitive data, the notion of differential privacy (DP) is used to introduce privacy by bounding the effect that small changes in the input data set can have on the output. This can be formally captured as follows. A protocol Π is said to offer (ϵ, δ) -differential privacy if, for all input sets I and I' that differ in a single entry and all possible subset of outcomes O, the following holds:

$$Pr[\Pi(I) \in O] \leq \exp(\epsilon) \cdot Pr[\Pi(I') \in O] + \delta$$

One way to achieve DP is to add Laplacian noise to the protocol output, as defined in [23]. The sensitivity of a protocol Π is defined as $\Delta = \max_{\forall I, I'} |\Pi(I) - \Pi(I')|$. The additive noise is calibrated as $\mathsf{Laplace}(\Delta/\epsilon)$, where $\mathsf{Laplace}(b)$ is a random variable from the Laplace distribution with a scale b and density $\mathsf{Laplace}(x; b) = \frac{1}{2b} \exp\left(-\frac{|x|}{b}\right)$. In order to achieve DP, we rely on the central model of DP, owing to the need for high accuracy, as done in [11], as opposed to the alternatives of local and the shuffle models. Since the central

1. Note that since our protocol for private heavy hitters treats both shuffle and sort protocols as black boxes, it can be instantiated with any efficient alternative. With respect to Quicksort, we note that to ensure optimal performance, the entries to be sorted can be made free of duplicates. For this, each input of length ℓ-bits can be extended to $\ell + k$ bits by appending random bits towards the least significant bit. This would randomize the inputs to mimic a uniform distribution.

model assumes a trusted server applying DP on clear text data, we instead simulate the same in a distributed setting via MPC, as in [11].

3 Static Setting

The process of identifying the τ-heavy hitters in the static setting comprises the following three phases:

1. *Input sharing and consistency check:* Each of the $\mathcal{N}$ clients secret-share their inputs among the servers. To ensure that a malicious client does not share malformed inputs, the servers verify the consistency of the received shares and discard malformed inputs, if any.
2. *Computation of private heavy-hitters:* The servers execute an MPC protocol on the $\mathcal{N}$ secret-shared inputs to compute the set of τ-heavy hitters (in secret-shares).
3. *Output Reconstruction:* Servers reconstruct the set of τ-heavy hitters towards the intended recipient.

Since the first and last step is dependent on the underlying MPC, we refer an interested reader to [14] for the details. We next discuss ideal functionality $\mathcal{F}_{\mathsf{PHH}}$ for computing τ-heavy hitters and protocol Π_{PHH}, which securely realizes it.

3.1 Secure computation of τ-heavy hitters

Let $\mathbf{a}$ denote an $\mathcal{N}$-sized vector with $\mathbf{a}[i]$ denoting the ℓ-bit input string of client i. $\mathcal{F}_{\mathsf{PHH}}$ (Fig. 7) takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathbf{a}$, which comprises $[\![\cdot]\!]^{\mathbf{B}}$-shares of each element in $\mathbf{a}$. It outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathbf{h}$, where $\mathbf{h}[i] = 1$ if $\mathbf{a}[i]$ is a heavy hitter and i is the index of the string's first occurrence in $\mathbf{a}$ (i.e., $\forall j < i, \mathbf{a}[j] \neq \mathbf{a}[i]$); $\mathbf{h}[i] = 0$ otherwise.

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

Let $\mathcal{S}$ denote the ideal world adversary. $\mathcal{F}_{\mathsf{PHH}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of elements in the $\mathcal{N}$-sized vector $\mathbf{a}$ and proceeds as follows.

– Reconstruct elements in input $\mathbf{a}$. Generate $\mathcal{N}$-sized vectors $\mathbf{h}, \mathbf{t}$, with all entries initialized to 0.

– For $i = 1$ to $\mathcal{N}$,

  - Count number of occurrences of $\mathbf{a}[i]$ and store it in $\mathbf{count}[i]$.
  - If $\mathbf{count}[i] \geq \tau$ and $\mathbf{a}[i] \neq \mathbf{a}[j]$ for $j < i$, set $\mathbf{h}[i] = 1$.

– Generate $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathbf{a}$ and $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathbf{h}$, and send $(\mathsf{Output}, [\![\mathbf{a}]\!]^{\mathbf{B}}_s, [\![\mathbf{h}]\!]^{\mathbf{B}}_s)$ to $P_s \in \mathcal{P}$.

Fig. 7: Ideal functionality for computing private heavy hitters

We next give a high-level overview of the secure protocol for Π_{PHH}. Note that unlike prior protocols such as Poplar [5], our goal is to design a secure protocol that does not leak any information other than the strings identified as heavy hitters. Hence, all the steps described next will be performed on $[\![\cdot]\!]$-shared (or $[\![\cdot]\!]^{\mathbf{B}}$-shared) values. An intuitive way to identify the heavy hitters is to securely sort the input $\mathbf{a}$, which ensures all the duplicate entries in $\mathbf{a}$ occur together. In the sorted $\mathbf{a}$, each unique occurrence of a string is tagged with a 1, while all the duplicates that follow the first occurrence of the string are tagged with a 0. It is evident that the difference of indices of consecutive 1s gives the count of each string. All those strings whose count exceeds the threshold τ are then marked as heavy hitters. This process is depicted in Fig. 8. While secure sorting can be accomplished in $\mathcal{O}(\log \mathcal{N})$ rounds of interaction [24], computing the count requires a linear traversal of $\mathbf{a}$, thereby requiring $\mathcal{O}(\mathcal{N})$ rounds. Hence, to ensure that determining the count of strings is not the bottleneck, departing from the $\mathcal{O}(\mathcal{N})$-round solution, we introduce novel approaches to efficiently compute the same, as described next.

We observe that we can avoid the linear scan if there is a way to ensure that the elements of $\mathbf{a}$ tagged with a 1, together with their indices, appear first, followed by all the elements and their corresponding indices that are tagged with a 0. This is because such an ordering allows calculating the distance between the consecutive 1s in the sorted $\mathbf{a}$ without requiring a linear scan to identify the last seen 1, as shown in Fig. 8. Thus, the difference of indices of consecutive locations which hold a 1 can be calculated in parallel, in constant rounds, to determine the count.

Algorithm Heavy hitter

Input: An $\mathcal{N}$-sized vector $\mathbf{a}$, where each element of $\mathbf{a}$ represents a client's ℓ-bit input string, and $\mathcal{N}$ denotes the number of clients.

Output: Reordered $\mathbf{a}$, and an $\mathcal{N}$-sized Boolean vector $\mathbf{h}$ corresponding to (reordered) $\mathbf{a}$ such that

$$\mathbf{h}[i] = \begin{cases} 1, & \text{if } \mathbf{a}[i] \text{ is a heavy hitter and for } j < i, \mathbf{a}[j] \neq \mathbf{a}[i] \\ 0, & \text{otherwise} \end{cases}$$

Protocol:

```
1. Sort elements in $\mathbf{a}$.
2. For $j = 1$ to $\mathcal{N}$, initialize $\mathbf{t}[j] = 0$, $\mathbf{h}[j] = 0$, and $\mathbf{index}[j] = j$.
3. Populate elements in $\mathbf{t}$ by tagging duplicates in sorted $\mathbf{a}$ with a 0 and non-duplicates with 1, as follows.
   $\mathbf{t}[1] = 1;\ \text{for } j = 2 \text{ to } \mathcal{N}:\ \mathbf{t}[j] = \begin{cases} 0, & \text{if } \mathbf{a}[j] = \mathbf{a}[j-1] \\ 1, & \text{otherwise} \end{cases}$
4. Compact the elements in $\mathbf{a}, \mathbf{t}, \mathbf{index}$ based on the 0/1 tags in $\mathbf{t}$.
5. For $j = 1$ to $\mathcal{N}$, mark $\mathbf{a}$ as a $\tau$-heavy hitter, i.e., set $\mathbf{h}[j] = 1$ if $\mathbf{t}[j] = 1$, AND
   – $\mathbf{t}[j+1] = 1$ and $\mathbf{index}[j+1] - \mathbf{index}[j] \geq \tau$, OR
   – $\mathbf{t}[j+1] = 0$, and $\mathcal{N} - \mathbf{index}[j] + 1 \geq \tau$,
6. Output $\mathbf{a}$ and $\mathbf{h}$.
```

Fig. 9: Computation of heavy hitters

Consequent to the above discussion, our goal is to obtain a compacted ordering of the tags (together with the corresponding entries in $\mathbf{a}$ and their indices) such that all the 1s appear first followed by all the 0s. Further, this should be realized with sub-linear round complexity. Moreover, to ensure correct computation of the count, it is required that this compaction be stable, i.e., the relative ordering of 1 tags in the input should be preserved in the compacted order. We abstract this task via the ideal functionality $\mathcal{F}_{\mathsf{Compact}}$. This functionality takes as input $[\![\cdot]\!]$-shares of a vector $\mathbf{t}$ of tags and shares of the corresponding vector of payloads $\mathbf{p}$ (in this case, elements in $\mathbf{a}$ and the associated indices). It outputs $[\![\cdot]\!]$-shares of the compacted vector $\mathbf{t_c}$ together with the shares of the corresponding payloads ordered as per $\mathbf{t_c}$. Moreover, we define $\mathcal{F}_{\mathsf{Compact}}$ to be stable if for every i, j entries in $\mathbf{t}$ that get mapped to i', j' in $\mathbf{t_c}$, where $i < j$, $\mathbf{t}[i] = \mathbf{t}[j]$, it holds that $i' < j'$.

A simple solution to realize $\mathcal{F}_{\mathsf{Compact}}$ securely with

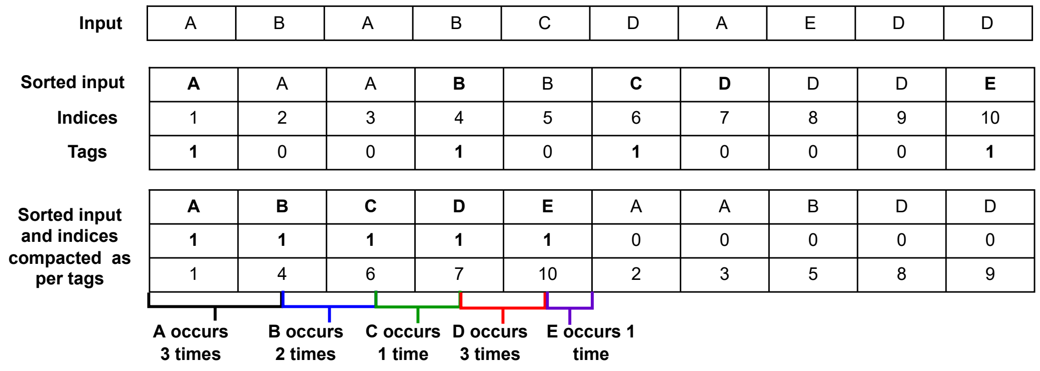

Fig. 8: Toy example of PHH computation

sublinear round complexity is to perform sorting (via a secure stable sort protocol) using elements in $\mathbf{t}$ as the keys. Although this enables an $\mathcal{O}(\log \mathcal{N})$ round protocol, eliminating the $\mathcal{O}(\mathcal{N})$ overhead, we ask the question: can we do better? We answer this affirmatively in §3.2 where we design a secure and stable compaction protocol that requires constant rounds in the $\mathcal{F}_{\mathsf{Shuffle}}$-hybrid model (defined in §2). Thus, given a secure protocol to efficiently realize $\mathcal{F}_{\mathsf{Compact}}$, we can determine the private heavy hitters in $\mathcal{O}(\log(\mathcal{N}))$ round complexity, where the complexity is dominated by that of secure sorting. Our algorithm for computing heavy hitters appears in Fig. 9.

Note in steps 3, 5 in Fig. 9, it is important to ensure that no information about whether a 0 or a 1 is assigned to $\mathbf{t}[j], \mathbf{h}[j]$, respectively, should be leaked. Hence, this assignment task is performed obliviously using the Π_{Select} protocol (see Table 3 for description). Moreover, checking the equality of elements as well as comparison operations are performed securely using Π_{Eq} and Π_{Cmp} protocols (see Table 3 for description). The secure version of the algorithm appears in Fig. 10. Informally, the security of our protocol follows from the security of the underlying primitives.

Lemma 3.1. *The protocol, Π_{PHH} (Fig. 10) securely realizes the functionality $\mathcal{F}_{\mathsf{PHH}}$ (Fig. 7) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Compact}})$-hybrid model when instantiated with the MPC of [14].*

Having computed the $[\![\cdot]\!]^{\mathbf{B}}$-shares of the output via Π_{PHH}, during output reconstruction one should not simply reconstruct $\mathbf{h}$ and the corresponding entries in $\mathbf{a}$ when $\mathbf{h}[j] = 1$, for $j = 1$ to $\mathcal{N}$. This is because of a subtle leak that may be possible owing to the reordering of the elements in $\mathbf{a}$ during the run of the protocol (where all unique elements appear first, followed by groups of duplicates). Elaborately, reconstructing only those entries in $\mathbf{a}$ which have a 1 in $\mathbf{h}$ may leak information about the existence and identity of non-heavy hitter strings that lie between two heavy hitter strings. To understand this with an example, consider the run of the protocol as illustrated in Fig. 8. If $\tau = 2$, then A, B, D constitute τ heavy hitters. Since $\mathbf{a}$ is ordered such that all unique elements appear together followed by duplicates, reconstructing these entries in $\mathbf{a}$ leaks information that there exists a string between B and D, and this string is not a heavy hitter. Since it is common knowledge that C is the only such string, an adversary can learn that C was supplied as input by one or more clients and that there were not sufficiently many clients with C as input to make it a heavy hitter. To avoid leaking such information, it is required to break the structure in $\mathbf{a}$ before reconstruction can begin. Hence, we securely shuffle $\mathbf{a}$ as well as $\mathbf{h}$ before reconstruction begins.

Protocol Π_{PHH}

Input: $[\![\cdot]\!]^{\mathbf{B}}$-shares of an $\mathcal{N}$-sized vector $\mathbf{a}$, where each element of $\mathbf{a}$ represents a client's ℓ-bit input string, and $\mathcal{N}$ denotes the number of clients.
Output: $[\![\cdot]\!]^{\mathbf{B}}$-shares of reordered $\mathbf{a}$, and $[\![\cdot]\!]^{\mathbf{B}}$-shares of an $\mathcal{N}$-sized Boolean vector $\mathbf{h}$ corresponding to (reordered) $\mathbf{a}$ such that

$$\mathbf{h}[i] = \begin{cases} 1, & \text{if } \mathbf{a}[i] \text{ is a heavy hitter and for } j < i, \mathbf{a}[j] \neq \mathbf{a}[i] \\ 0, & \text{otherwise} \end{cases}$$

Protocol:

1. Invoke $\mathcal{F}_{\mathsf{Sort}}$ to securely sort the $[\![\cdot]\!]^{\mathbf{B}}$-shared elements in $\mathbf{a}$.
2. For $j = 1$ to $\mathcal{N}$, initialize $[\![\mathbf{t}[j]]\!] = [\![0]\!]$, $[\![\mathbf{h}[j]]\!]^{\mathbf{B}} = [\![0]\!]$, and $[\![\mathbf{index}[j]]\!] = [\![j]\!]$.
3. Populate elements in $\mathbf{t}$ by tagging duplicates in sorted $\mathbf{a}$ with a 0 and non-duplicates with 1, as follows.
   - $[\![\mathbf{t}[1]]\!] = [\![1]\!]$
   - For $j = 2$ to $\mathcal{N}$:
     - $[\![\mathsf{b}_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathbf{a}[j]]\!], [\![\mathbf{a}[j-1]]\!])$
     - $[\![\mathbf{t}[j]]\!] = \Pi_{\mathsf{Select}}([\![0]\!], [\![1]\!], [\![\mathsf{b}_1]\!]^{\mathbf{B}})$
4. Invoke $\mathcal{F}_{\mathsf{Compact}}$ to securely compact the shared elements in $\mathbf{a}, \mathbf{t}, \mathbf{index}$ based on the 0/1 tags in $\mathbf{t}$.
5. For $j = 1$ to $\mathcal{N}$:
   - $[\![\mathsf{b}_{21}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathbf{t}[j] + \mathbf{t}[j+1]]\!], [\![2]\!])$
   - $[\![\mathsf{b}_{22}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Cmp}}([\![\tau]\!], [\![\mathbf{index}[j+1] - \mathbf{index}[j] + 1]\!])$
   - $[\![\mathsf{b}_{31}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathbf{t}[j] + \mathbf{t}[j+1]]\!], [\![1]\!])$
   - $[\![\mathsf{b}_{22}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Cmp}}([\![\tau]\!], [\![\mathcal{N} - \mathbf{index}[j] + 2]\!])$
   - $[\![\mathsf{b}_{2}]\!]^{\mathbf{B}} = \Pi_{\mathsf{mult}}([\![\mathsf{b}_{21}]\!]^{\mathbf{B}}, [\![\mathsf{b}_{22}]\!]^{\mathbf{B}})$
   - $[\![\mathsf{b}_{3}]\!]^{\mathbf{B}} = \Pi_{\mathsf{mult}}([\![\mathsf{b}_{31}]\!]^{\mathbf{B}}, [\![\mathsf{b}_{32}]\!]^{\mathbf{B}})$
   - $[\![\mathsf{b}_{4}]\!]^{\mathbf{B}} = \Pi_{\mathsf{mult}}([\![1 \oplus \mathsf{b}_{2}]\!]^{\mathbf{B}}, [\![1 \oplus \mathsf{b}_{3}]\!]^{\mathbf{B}})$
   - $[\![\mathbf{h}[j]]\!]^{\mathbf{B}} = \Pi_{\mathsf{Select}}([\![1]\!]^{\mathbf{B}}, [\![0]\!]^{\mathbf{B}}, [\![1 \oplus \mathsf{b}_{4}]\!]^{\mathbf{B}})$
6. Output $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathbf{a}$ and $\mathbf{h}$.

Fig. 10: Protocol for private heavy hitters

3.2 Secure stable compaction

Recall that the ideal functionality for stable compaction $\mathcal{F}_{\mathsf{Compact}}$ (as described in Fig. 11) takes as input $[\![\cdot]\!]$-shares of an $\mathcal{N}$-sized vector $\mathbf{t}$ of tags (each element of $\mathbf{t}$ is either 0 or 1), and $[\![\cdot]\!]$-shares (or $[\![\cdot]\!]^{\mathbf{B}}$-shares) of the corresponding vector of payloads $\mathbf{p}$. It outputs $[\![\cdot]\!]$-shares of the compacted vector $\mathbf{t_c}$ together with the $[\![\cdot]\!]$-shares (or $[\![\cdot]\!]^{\mathbf{B}}$-shares) of the corresponding payloads ordered as per $\mathbf{t_c}$ such that the following hold:

– All 1s appear before all 0s in the compacted vector $\mathbf{t_c}$.

– For every i, j entries in $\mathbf{t}$ that get mapped to i', j' in $\mathbf{t_c}$, where $i < j$, $\mathbf{t}[i] = \mathbf{t}[j]$, it must hold that $i' < j'$. This property ensures that compaction is stable. The mapping from $\mathbf{t}$ to $\mathbf{t_c}$ is reflected in the output $\mathbf{p}$ too, i.e., for every i' mapped to i $\mathbf{p}[i']$ is defined to be $\mathbf{p}[i]$.

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

Let $\mathcal{S}$ denote the ideal world adversary. $\mathcal{F}_{\mathsf{Compact}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of elements in the $\mathcal{N}$-sized vector $\mathbf{t}$ where each element is a 0 or 1, and $[\![\cdot]\!]$-shares of payloads associated with each entry in $\mathbf{t}$, denoted by the vector $\mathbf{p}$. $\mathcal{F}_{\mathsf{Compact}}$ proceeds as follows.

– Reconstruct the inputs $\mathbf{t}$ and $\mathbf{p}$.

– Construct $\mathbf{t_c}$ by reordering elements of $\mathbf{t}$ while adhering to the following.

- All 1s appear before all 0s.
- If entries at location i and j in $\mathbf{t}$ where $i < j$ and $\mathbf{t}[i] = \mathbf{t}[j]$ get mapped to locations i' and j', respectively, in $\mathbf{t_c}$, then $i' < j'$.

– Use the mapping from $\mathbf{t}$ to $\mathbf{t_c}$ to reorder elements in $\mathbf{p}$.

– Generate $[\![\cdot]\!]$-shares of $\mathbf{t_c}$ and $\mathbf{p}$ and send $(\mathsf{Output}, [\![\mathbf{t_c}]\!]_s, [\![\mathbf{p}]\!]_s)$ to $P_s \in \mathcal{P}$.

Fig. 11: Ideal functionality for stable compaction

The high level idea to obtain the compacted vector $\mathbf{t_c}$ from $\mathbf{t}$, is to assign each element in $\mathbf{t}$ a unique label between 1 to $\mathcal{N}$, such that the labels assigned to all the 1s are smaller than the labels assigned to the 0s. Let $\mathbf{label}[i]$ denote the label assigned to $\mathbf{t}[i]$. Sorting the elements in $\mathbf{t}$ based entries in $\mathbf{label}$ ensures that all the 1s appear before the 0s, thereby generating the compacted vector. Further, to ensure that the compaction is stable, labels assigned to 1s (and 0s, respectively) should be such that if $\mathbf{t}[i] = \mathbf{t}[j]$ and $i < j$, then the $\mathbf{label}[i] < \mathbf{label}[j]$. Note that while the assignment of labels should be done in secret shared format, the sorting of $\mathbf{t}$ based on the labels can be performed after revealing the labels on clear. This allows us to perform the sort operation non-interactively. Elaborately, the secret shared $\mathbf{label}$ is reconstructed, followed by locally sorting it. In the process, the corresponding shares of $\mathbf{t}$ are also reordered locally. Thus, our goal reduces to generating $\mathbf{label}$, as described above, with constant round complexity.

Before we describe our solution, we note that the recent work of [12] that addresses this problem has the following drawbacks: (i) it requires an $\mathcal{O}(\mathcal{N})$ round complexity, (ii) it assumes that the number of elements which are a 1 in $\mathbf{t}$, denoted by count_1, is public knowledge. We overcome both these drawbacks in our solution. We begin with briefly explaining the solution in [12] to help better place ours. The protocol of [12] proceeds by sequentially scanning $\mathbf{t}$ such that each time $\mathbf{t}[i] = 1$, it increments a counter (initialized to 0) and assigns it as the label to the current 1-entry in $\mathbf{t}$. Similar is the case while assigning labels to the 0-entry, except that the counter for 0s is initialized to count_1, which ensures that the 0s get a label higher than the 1s. We depart from this approach and devise a method to not only assign labels to all the elements in constant rounds but also do not require public knowledge of count_1. We proceed as follows.

We define two vectors, $\mathbf{c_1}$ and $\mathbf{c_0}$ where, $\mathbf{c}_b$ for $b \in \{0, 1\}$ tracks the number of b's present in $\mathbf{t}$ until index i. Assuming that $\mathbf{c_1}$ and $\mathbf{c_0}$ are available (in secret-shared format), $\mathbf{label}[i]$ can be set as $\mathbf{c_1}[i]$ if $\mathbf{t}[i] = 1$. On the other if $\mathbf{t}[i] = 0$, then $\mathbf{label}[i]$ can be set as $\mathbf{c_1}[\mathcal{N}] + \mathbf{c_0}[i]$. Observe here that the offset $\mathbf{c_1}[\mathcal{N}]$ ensures that all 0s get assigned a label that is higher than that of 1s. This ensures that 0s appear after the 1s when $\mathbf{t}$ is sorted based on the labels. Since $\mathbf{c_1}$ is only available in shares, it ensures that we do not leak any information about the number of 1s, unlike in the protocol of [12]. Further, note that this assignment of labels does not require a *sequential* scan of $\mathbf{t}$ and can be performed in parallel for each entry in $\mathbf{t}$, when $\mathbf{c_1}$ and $\mathbf{c_0}$ are available. Such a parallel assignment of labels is not possible in the protocol of [12] since they assign the labels based on a counter (that keeps track of the number of 1s seen so far) which is updated sequentially.

Consequent to the above discussion, we complete the description of our constant round protocol by discussing how $\mathbf{c_1}$ and $\mathbf{c_0}$ can be populated in constant rounds. In fact, we note that these can be populated without any interaction, as follows. Observe that $\mathbf{c_1}[i]$ can be computed as $\sum_{j=1}^{i} \mathbf{t}[j]$. Moreover, since $\mathbf{t}$ has either 1s or 0s, $\mathbf{c_0}[i]$ can be computed as $i - \mathbf{c_1}[i]$. Since computing $\mathbf{c_1}[i]$ and $\mathbf{c_0}[i]$ involves linear operations, it can be performed non-interactively, given any linear secret sharing scheme.

Algorithm Compaction

Input: An $\mathcal{N}$-sized vector $\mathbf{t}$ whose elements are either 0 or 1, and $\mathbf{p}$ which comprise payload associated with each element of $\mathbf{t}$.
Output: Compacted vector $\mathbf{t_c}$ where all 1s appear before 0s, and $\mathbf{p}$ with elements reordered as per $\mathbf{t_c}$.
Protocol:

```
1. Construct $\mathcal{N}$-sized vectors $\mathbf{c_0}$ and $\mathbf{c_1}$ to count number
   of 0s and 1s in $\mathbf{t}$ as follows.
   – $\mathbf{c_1}[1] = \mathbf{t}[1]$,   $\mathbf{c_0}[1] = 1 - \mathbf{c_1}[1]$
   – For $j = 2$ to $\mathcal{N}$:
     - $\mathbf{c_1}[j] = \mathbf{c_1}[j-1] + \mathbf{t}[j]$
     - $\mathbf{c_0}[j] = j - \mathbf{c_1}[j]$
2. Construct $\mathcal{N}$-sized vector $\mathbf{label}$ as follows.
   $\mathbf{label}[j] = \begin{cases} \mathbf{c_0}[j] + \mathbf{c_1}[\mathcal{N}], & \text{if } \mathbf{t}[j] = 0 \\ \mathbf{c_1}[j], & \text{otherwise} \end{cases}$
3. Shuffle the elements in $\mathbf{p}, \mathbf{t}, \mathbf{label}$ using the same
   random permutation.
4. For $i = 1$ to $\mathcal{N}$, set
   – $\mathbf{p}[\mathbf{label}[i]] = \mathbf{p}[i]$
   – $\mathbf{t_c}[\mathbf{label}[i]] = \mathbf{t}[i]$
```

Fig. 12: Stable compaction

To summarize, the protocol begins by generating $\mathbf{c_1}, \mathbf{c_0}$, non-interactively. This is followed by the generation of $\mathbf{label}$, which can be performed in constant rounds (independent of $\mathcal{N}$). Following this, we can reconstruct the entries in $\mathbf{label}$, based on which we can sort the elements in $\mathbf{t}$ together with their respective payloads. A subtle thing to note here is that reconstructing the entries in $\mathbf{label}$ can leak information. This is because the distribution of 1s and 0s in $\mathbf{t}$ induces a structure on the entries in $\mathbf{label}$. For example, assuming

p	A	A	A	B	B	C	D	D	D	E
t	1	0	0	1	0	1	1	0	0	1

c₀	0	1	2	2	3	3	3	4	5	5
c₁	1	1	1	2	2	3	4	4	4	5
label	1	6	7	2	8	3	4	9	10	5

p, **t**, **label** after random shuffle										
	B	E	A	D	B	C	D	A	D	A
	0	1	1	0	1	1	0	0	1	0
	8	5	1	10	2	3	9	7	4	6

p, **t**, sorted based on reconstructed label										
	A	B	C	D	E	A	A	B	D	D
	1	1	1	1	1	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10

Fig. 13: A toy run of compaction protocol

Protocol Π_{Compact}

Input: $[\![\cdot]\!]$-shares of an $\mathcal{N}$-sized vector $\mathbf{t}$ whose elements are either 0 or 1, $[\![\cdot]\!]$-shares of $\mathbf{p}$ which comprise payload associated with each element of $\mathbf{t}$.
Output: $[\![\cdot]\!]$-shares of compacted vector $\mathbf{t_c}$ where all 1s appear before 0s, and $[\![\cdot]\!]$-shares of $\mathbf{p}$ with elements reordered as per $\mathbf{t_c}$.
Protocol:

1. Construct $\mathcal{N}$-sized $[\![\cdot]\!]$-shared vectors $\mathbf{c_0}$ and $\mathbf{c_1}$ to count number of 0s and 1s in $\mathbf{t}$ as follows.
   – $[\![\mathbf{c_1}[1]]\!] = [\![\mathbf{t}[1]]\!]$, $\quad [\![\mathbf{c_0}[1]]\!] = [\![1]\!] - [\![\mathbf{c_1}[1]]\!]$
   – For $j = 2$ to $\mathcal{N}$:
     - $[\![\mathbf{c_1}[j]]\!] = [\![\mathbf{c_1}[j-1]]\!] + [\![\mathbf{t}[j]]\!]$
     - $[\![\mathbf{c_0}[j]]\!] = [\![j]\!] - [\![\mathbf{c_1}[j]]\!]$
2. Construct $\mathcal{N}$-sized $[\![\cdot]\!]$-shared vector **label** as follows.

   – $[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}\left([\![\mathbf{t}[j]]\!], [\![0]\!]\right)$
   – $[\![\mathbf{label}[j]]\!] = \Pi_{\mathsf{Select}}\left([\![\mathbf{c_1}[j]]\!], [\![\mathbf{c_0}[j] + \mathbf{c_1}[\mathcal{N}]]\!], [\![\mathsf{b}]\!]^{\mathbf{B}}\right)$
3. Invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to securely shuffle the $[\![\cdot]\!]$-shared elements in $\mathbf{p}, \mathbf{t}, \mathbf{label}$ using the same random permutation, and reconstruct elements in **label**.
4. For $i = 1$ to $\mathcal{N}$, set
   – $[\![\mathbf{p}[\mathbf{label}[i]]]\!] = [\![\mathbf{p}[i]]\!]$
   – $[\![\mathbf{t_c}[\mathbf{label}[i]]]\!] = [\![\mathbf{t}[i]]\!]$

Fig. 14: Protocol for secure stable compaction

that there is at least one 1 in $\mathbf{t}$, if the reconstructed value $\mathbf{label}[1] = 1$, it reveals that the first entry in $\mathbf{t}$ is a 1. Else if $\mathbf{label}[1] > 1$, it leaks the number of 1s in $\mathbf{t}$. This leakage is evident from the example given in Fig. 13. To avoid such leakage, prior to reconstructing **label**, we securely shuffle the entries in $\mathbf{label}, \mathbf{t}$ and the respective payloads. This allows to break any association among the entries in **label** and $\mathbf{t}$. In this way, the round complexity of our protocol depends on that of the shuffle protocol. For the 3PC setting considered in our work, we rely on the secure shuffle protocol of [13], whose round complexity is independent of $\mathcal{N}$. Thus, the overall complexity of our compaction protocol is also independent of $\mathcal{N}$. The algorithm for compaction appears in Fig. 12 while its secure variant appears in Fig. 14. Informally, the security of the protocol follows from the security of the underlying primitives. Additionally, note that the reconstructed **label** is always a random sequence of elements between 1 to $\mathcal{N}$ since it is the output of shuffle. Hence, this does not leak any additional information.

Lemma 3.2. *The protocol, Π_{Compact} (Fig. 14) securely realizes the functionality $\mathcal{F}_{\mathsf{Compact}}$ (Fig. 11) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Shuffle}})$-hybrid model when instantiated with the MPC of [14].*

4 Streaming Setting

This setting, unlike the static case, considers client inputs arriving at discrete time steps. Due to this sequential nature in which inputs arrive, heavy hitters must be identified at intervals by considering all the inputs seen so far. To design efficient solutions that can scale well with streaming inputs, the objective is to identify the top-K most popular strings among the input stream. Thus, the streaming setting is parameterized by K that bounds the total number of strings that can be output as a heavy-hitter, rather than identifying all strings that occur more than τ times, as done in the static setting. Note that recomputing heavy hitters at intervals can compromise client privacy. For instance, consider the computation of heavy hitters at time steps t and t', where the input stream in the latter additionally comprises a new client's input. In such a scenario, the outputs at t and t' can leak information about the input of the new client. Thus, to guarantee additional privacy for clients when processing streaming inputs, the use of *differential privacy* (DP) has been widely adopted [3], [11], [25], [26].

The work of [11] provides the state-of-the-art secure protocols for computing the heavy hitters in the streaming setting. Among the various cleartext techniques that have been used to detect heavy-hitters, [11] relies on a counter-based approach which is known to be favorable for processing a stream of inputs. This approach, where a set of counters are updated each time a new input arrives, is known to provide the best space and time complexity, along with accuracy. Specifically, [11] considers the popular *Misra-Gries* algorithm [27] that cleverly approximates the output, even when the input domain is unknown. The Misra-Gries algorithm maintains a vector $\mathbf{v}$ of size K to store the inputs that could potentially be identified as the heavy-hitters as well as their approximate counts in $\mathbf{c}$. Each time a new input arrives, say x, the following steps are carried out. If an entry for the input is already present in $\mathbf{v}$ (i.e., $\mathsf{x} \in \mathbf{v}$), then its corresponding count in $\mathbf{c}$ is incremented. However, in case x is not present in $\mathbf{v}$, there are two possibilities. If $\mathbf{v}$ has empty slots, then x is inserted in one of the empty slots and its count in $\mathbf{c}$ is set to 1. On the other hand, if $\mathbf{v}$ has no free slots, the count of all entries in $\mathbf{c}$ is decremented, and all those entries that have zero count are removed from $\mathbf{v}$. Following these sequence of steps on a stream of inputs is known to provide the guarantee that when evaluated on $\mathcal{N}$ inputs while maintaining K counters, the estimated frequency $f'[i]$ of each input i satisfies the following bound: $0 \leq f[i] - f'[i] \leq \mathcal{N}/(K+1)$. Here, $f[i]$ denotes the true frequency of an input i and is defined as the ratio of the number of times it occurs (count) to the total number of inputs seen so far. We refer readers to [28] for the correctness of the algorithm. Further, to provide $(\Delta\epsilon, \delta)$-differential privacy, [11] modifies the algorithm to reveal noisy counts that exceed a predetermined threshold τ', defined in [29] as

$$\tau' = 1 - \Delta \log\left(2 - 2(1-\delta)^{\frac{1}{\Delta}}\right)/\epsilon$$

Here, Δ denotes the maximum contribution of each input string to its overall count and hence is set to 1. The ideal functionality for the same is given in Fig. 15.

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

$\mathcal{F}_{\mathsf{SPHH}}$ maintains a vector $\mathbf{v}$ of size K to store the heavy-hitters so far and their corresponding count in $\mathbf{c}$.
Let $\mathcal{S}$ denote the ideal world adversary. $\mathcal{F}_{\mathsf{SPHH}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of new input, say x, and proceeds as follows.

```
– Reconstruct $\mathsf{x}$.
– If $\mathsf{x} \in \mathbf{v}$ at index $i$, then increment its count, $\mathbf{c}[i]$.
– Else if there is an empty slot in $\mathbf{v}$, then set $\mathbf{v}[i] = \mathsf{x}$ where
  $i$ denotes the first empty slot, and set its count as $\mathbf{c}[i] = 1$.
– Else decrement count of each entry in $\mathbf{c}$.
//Ensuring differential privacy before output reconstruction:
– For each entry $\mathbf{v}[i]$
    - Add Laplacian noise Laplace($\Delta/\epsilon$) to its count $\mathbf{c}[i]$.
    - If $\mathbf{c}[i] < \tau'$, set $\mathbf{v}'[i] = \bot$ else set $\mathbf{v}'[i] = \mathbf{v}[i]$.
- Output $[\![\cdot]\!]$-shares of entries in $\mathbf{v}'$.
```

Fig. 15: Ideal functionality for streaming private heavy hitters

When designing a secure variant for Misra-Gries, the following aspects must be considered. We note that the algorithm does not qualify to be *data-oblivious*. This is evident from the fact that the following information is leaked by the control flow of the algorithm - (i) the number of valid entries in $\mathbf{v}$, (ii) whether the input being processed is present in $\mathbf{v}$ or not, (iii) the entry in $\mathbf{v}$ whose count in $\mathbf{c}$ is incremented (if present), (iii) whether $\mathbf{v}$ has empty slots or not and (iv) the entries in $\mathbf{v}$ that fail to satisfy the differentially private threshold τ'. Thus, one must ensure that none of this information is leaked in its secure variant.

To ensure data obliviousness, [11] designs a secure variant of Misra-Gries such that each branch of *if-else* is evaluated, however, the use of flag bits ensure that only the operations in the correct branch is evaluated. In this way, information regarding which branch was evaluated is not leaked. Further, their protocol is optimized to ensure- (i) interactive operations such as multiplications are avoided where possible, (ii) a reduced number of *conditional swaps*[2], and (iii) avoid explicitly deleting entries with zero count as done in Misra-Gries, which additionally saves a few operations. We optimize their MPC protocol further to achieve improved efficiency and overall run time, as described next.

Algorithm Streaming heavy hitter

```
State:
//Information maintained throughout the protocol
  1. Vector $\mathbf{v}$ of size $K$ to store the heavy-hitters so far,
     such that each entry is initialized to $\bot$.
  2. Vector $\mathbf{c}$ of size $K$ to store count of each entry in $\mathbf{v}$,
     such that each entry is initialized to $0$.
Processing Client input
//Initiated when a new client arrives with its input $\mathsf{x}$ consisting of
an $\ell$-bit string
  1. Initialize the flags $found$ and $free$ to 0.
  2. For $i = 1$ to $K$
     - If the current entry $\mathbf{v}[i] = \mathsf{x}$, set the flag $found = 1$ and
       increment the count in $\mathbf{c}[i] = \mathbf{c}[i] + 1$.
     - Mark if current entry in $\mathbf{v}$ is empty by maintaining $\mathbf{e}$ as:
         $\mathbf{e}[i] = \begin{cases} 1, & \text{if } \mathbf{c}[i] < 1 \\ 0, & \text{otherwise} \end{cases}$
     - Initialize $\mathbf{index}[i] = i$
  3. Having populated the empty vector $\mathbf{e}$, set the $free$ flag
     as the dot-product of $\mathbf{e}$ and $\mathbf{1}$.
  4. Invoke $\Pi_{\mathsf{Compact}}$ to securely compact elements in array
     $\mathbf{e}$ with array $\mathbf{index}$ treated as its payload.
  5. Determine the location $j$ where $\mathsf{x}$ needs to be inserted
     in $\mathbf{v}$ if not already present as:
         $j = \begin{cases} \mathbf{index}[1], & \text{if } free = 1 \\ -1, & \text{otherwise} \end{cases}$
  6. Set the flag $dec = 1$ if both $found$ AND $free$ are not
     set to 1.
  7. For $i = 1$ to $K$
     - Update $\mathbf{v}[i] = \mathsf{x}$ if $i = j$.
     - Similarly, update the counts in $\mathbf{c}$ as:
         $\mathbf{c}[i] = \begin{cases} 1, & \text{if } i = j \\ \mathbf{c}[i] - dec, & \text{otherwise} \end{cases}$
Output reconstruction:
//Initiated when heavy hitters are to be found and requires as
input the Laplacian noise $\mathsf{r}$ and DP threshold $\tau'$
  1. For $i = 1$ to $K$
     - Add the Laplacian noise to each count value
       $\mathbf{c}[i] = \mathbf{c}[i] + r$.
     - Initialize the entry $\mathbf{v}'[i]$ to $\bot$ if $\mathbf{c}[i] < \tau'$ else set
       $\mathbf{v}'[i] = \mathbf{v}[i]$.
  2. Output $\mathbf{v}'$.
```

Fig. 16: Computation of private heavy hitters in streaming case

As described in Fig. 16, we begin by initializing $\mathbf{v}$ to $\bot$ and $\mathbf{c}$ to all 0s. When a client's input x arrives, the flag $found$ is set if x is present in $\mathbf{v}$, and its count is incremented in $\mathbf{c}$. We additionally maintain a vector $\mathbf{e}$ to track all those entries in $\mathbf{v}$ that correspond to empty entries, i.e., their count in $\mathbf{c}$ is less than 1. The flag $free$ is set if at least one entry in $\mathbf{e}$ is set, indicating a free slot. In order to later identify the index of a free slot where x can be inserted, we additionally maintain the $\mathbf{index}$ vector, where the i^{th} entry is initialized to i. This corresponds to steps (1-3) in Fig. 16. Since each

2. A conditional swap is nothing but *oblivious select* operation which takes three inputs and outputs either the first or the second input based on whether the third input is a 1 or a 0.

iteration of step (2) is independent of the other, they can be carried out in parallel. The parallel iterations, however, only allow identifying the presence or absence of a free slot rather than the index of one such slot. The protocol in [11] also suffers from the same issue. Hence, they resort to a sequential approach to identify the empty slot. Instead, we rely on the Π_{Compact} protocol to achieve this in constant rounds. Specifically, in step (4), we securely compact the array $\mathbf{e}$ while treating **index** as the payload. In this way, we are guaranteed that the first index in the compacted **index** corresponds to a free slot if the flag $free$ was set previously (step (2)). Next, to determine if x is to be inserted with count 1 or if the count of all entries is to be decremented, we use the flag dec. This flag is set if x was not found in $\mathbf{v}$ and if there were no free slots. In step (7), we update $\mathbf{v}$ and $\mathbf{c}$ depending on dec. Note again that each iteration in step (7) can be performed in parallel. Finally, we note that steps for

```
Protocol $\Pi_{\mathsf{SPHH}}$

State:
  1. $[\![\mathbf{v}]\!]$, which is initialized as $[\![\mathbf{v}[i]]\!] = \perp$, for $i = 1$ to $K$.
  2. $[\![\mathbf{c}]\!]$, which is initialized as $[\![\mathbf{c}[i]]\!] = 0$, for $i = 1$ to $K$.
Processing client input:
// Client inputs $[\![\mathsf{x}]\!]$ to the servers
  1. For $i = 1$ to $K$
     – $[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathbf{v}[i]]\!], [\![\mathsf{x}]\!])$
     – $[\![found]\!]^{\mathbf{B}} = [\![found]\!]^{\mathbf{B}} \oplus [\![\mathsf{b}]\!]^{\mathbf{B}}$
     – $[\![\mathbf{c}[i]]\!] = [\![\mathbf{c}[i]]\!] + \Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$
     – $[\![\mathbf{e}[i]]\!]^{\mathbf{B}} = \Pi_{\mathsf{Cmp}}([\![\mathbf{c}[i]]\!], [\![1]\!])$
     – $[\![\mathbf{index}[i]]\!] = [\![i]\!]$
  2. $[\![free]\!]^{\mathbf{B}} = \Pi_{\mathsf{dotp}}([\![\mathbf{e}]\!]^{\mathbf{B}}, [\![\mathbf{1}]\!]^{\mathbf{B}})$ //$\mathbf{1}$ denotes vector of all 1s
  3. Invoke $\Pi_{\mathsf{Compact}}$ to securely compact elements in array
     $\mathbf{e}$ with array index treated as its payload.
  4. $[\![j]\!] = \Pi_{\mathsf{Select}}([\![\mathbf{index}[1]]\!], [\![-1]\!], [\![free]\!]^{\mathbf{B}})$
  5. $[\![dec]\!]^{\mathbf{B}} = \overline{[\![found]\!]^{\mathbf{B}}} \cdot \overline{[\![free]\!]^{\mathbf{B}}}$
  6. $[\![dec]\!] = \Pi_{\mathsf{bit2A}}([\![dec]\!]^{\mathbf{B}})$
  7. For $i = 1$ to $K$
     – $[\![b]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![j]\!], [\![i]\!])$
     – $[\![\mathbf{v}[i]]\!] = \Pi_{\mathsf{Select}}([\![\mathsf{x}]\!], [\![\mathbf{v}[i]]\!], [\![b]\!]^{\mathbf{B}})$
     – $[\![c]\!] = [\![\mathbf{c}[i]]\!] - [\![dec]\!]$
     – $[\![\mathbf{c}[i]]\!] = \Pi_{\mathsf{Select}}([\![1]\!], [\![c]\!], [\![b]\!]^{\mathbf{B}})$
Output reconstruction:
// Servers have shares of Laplacian noise $[\![\mathsf{r}]\!]$ & DP threshold $[\![\tau']\!]$

  1. For $i = 1$ to $K$
     – $[\![\mathbf{c}[i]]\!] = [\![\mathbf{c}[i]]\!] + [\![\mathsf{r}]\!]$
     – $[\![b]\!]^{\mathbf{B}} = \Pi_{\mathsf{Cmp}}([\![\mathbf{c}[i]]\!], [\![\tau']\!])$
     – $[\![\mathbf{v}'[i]]\!] = \Pi_{\mathsf{Select}}([\![\perp]\!], [\![\mathbf{v}[i]]\!], [\![b]\!]^{\mathbf{B}})$
  2. Output $[\![\cdot]\!]$-shares of array $\mathbf{v}'$.
```

Fig. 17: Protocol to compute private heavy hitters when inputs are streaming

output reconstruction is executed only when the heavy hitters are to be identified. When doing so, to ensure differential privacy, the first step is to add noise (as done in [11]) to the count and output only those entries with count higher than a predetermined threshold. In order to ensure the state information remains unaffected, the output is updated and stored separately in $\mathbf{v}'$. In this way, rather than relying on a sequential method to identify the empty slot where the input can be inserted, we provide a solution where all steps can be performed in parallel. Thus, we describe a constant round protocol in comparison to the linear (in K) round protocol in [11]. We note that all of the steps described above are performed on secret shared values. The formal description of the secure variant is provided in Fig. 17. Informally, the security of the protocol follows from the security of the underlying primitives.

Lemma 4.1. *The protocol, Π_{SPHH} (Fig. 17) securely realizes the functionality $\mathcal{F}_{\mathsf{SPHH}}$ (Fig. 15) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Compact}})$-hybrid model when instantiated with the MPC of [14].*

5 Benchmarks

We analyze the performance of Vogue by considering various parameters and compare our static and streaming solutions against their state-of-the-art solution of Poplar [5] and [11], respectively. We benchmark the performance over WAN using n1-standard instances of Google Cloud with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors and 240GB of RAM Memory. The machines have a bandwidth of 2Gbps. The instances are located in Iowa(P_0), South Carolina (P_1) and Oregon (P_2). We rely on the 3-party computation of [14] for the underlying MPC. We benchmark [14] on MOTION2NX framework [30] since it is not publicly available. Our code accounts for multi-threading wherever possible. Moreover, due to the unavailability of code for [11], and for a fair comparison, we benchmark their protocol (streaming setting) over the MPC of [14].

We follow the standard practice of benchmarking the honest execution [14], [19], [21]. We consider the run time and communication of the protocols as the benchmark parameters. Since the MPC of [14] works in the preprocessing paradigm, we report both the online as well as the preprocessing cost when doing so. Recall that online time captures the response time of the system (which is the time taken from submission of the input, processing it, to the generation of output). We first analyze the performance of our protocols in static setting, followed by streaming setting.

5.1 Static Setting

We adhere to similar experimental choices as considered in [5] while reporting the performance of Vogue. To analyze server side efficiency, we first vary the input string length of clients for a fixed number of clients (= 100K) and report the performance in Table 4. In comparison to [5], Vogue has a response time that is up to 10× faster. Even when accounting for the overall time, Vogue significantly outperforms [5]. We also report the server-side performance for a varying number of clients in Table 5. Here, we observe Vogue is up to 6.1× faster. Our improvements can be attributed to our improved heavy hitters protocol whose round complexity is logarithmic in the number of clients as opposed that of [5], which is linear in the length of client's input string.

With respect to client-side efficiency, we report the computational overhead at the client for varying input string lengths in Table 6. As evident from Table 6, Vogue has up to 48× improvements over [5]. The improvement is due to the lightweight operations at the client, as opposed to distributed point function key generation required in [5].

In this way, Vogue outperforms Poplar [5] not only in terms of efficiency but also by providing better security (since Poplar only provides partial security against malicious servers). To summarize, computation of heavy hitters for $\tau = 100$ with 400K clients on 256-bit input strings takes around 86 minutes on Poplar, whereas it is around 14 minutes for us.

Client string length (bits)	Vogue		Poplar [5]
	Preproc.	Online	
128	9.09	94.35	511.70
256	9.12	183.17	1041.38
512	9.19	396.96	3978.47

TABLE 4: Static: Server side run time in seconds for varying client string lengths ($\tau = 900$, #Client=100K).

#Client	Vogue		Poplar [5]
	Preproc.	Online	
100K	9.13	183.17	1041.38
200K	9.99	382.37	2627.07
400K	10.33	850.21	5191.61

TABLE 5: Static: Server-side run time for varying number of clients on 256-bit input string ($\tau = 900$).

Client string length (bits)	Vogue	[5]
128	2.50	120.8
256	5.82	173.1
512	8.89	204.9

TABLE 6: Static: Client-side run time in ($\times 10^{-6}$) seconds for varying client string lengths.

5.2 Streaming Setting

The performance of our protocol for computing heavy hitters when inputs arrive in a streaming fashion, as well as that of [11] is dependent on the number of clients and K (where K denotes the top-K inputs to be identified). Hence, to capture improvements brought in by our protocol over that of [11], we report the performance comparison of the protocols by varying both these parameters. In Table 7, we report the performance by varying K. We observe that the run time of our protocol does not increase drastically, as compared to [11] with increasing K. Further, we observe improvements of up to $3.5\times$ for $K = 100$, which when extrapolated, will increase as K increases. This improvement is due to the optimizations introduced in our protocol, where, unlike in the case of [11], the round complexity of our protocol is independent of K.

K	Vogue		[11]	
	Preproc. (sec)	Online (min)	Preproc. (sec)	Online (min)
4	9.65	11.46	9.33	9.64
8	9.16	11.49	10.62	10.72
16	9.47	11.46	12.84	13.04
32	9.43	11.43	17.39	18.65
64	9.84	11.33	23.57	28.80
100	9.36	11.48	27.46	40.94

TABLE 7: Streaming: Server-side run time for varying K for #Clients $= 300$.

In Table 8, we report the performance comparison for varying number of clients, where the computation of heavy hitters is performed after a stream of the reported number of client inputs arrives. As expected, in this scenario, the run time of Vogue is better than that of [11]. Further, the factor improvement in the run time of our protocol is observed to increase gradually. This is because although the number of clients has the same impact on both protocols, the gain in the run time of our protocol due to its independence from K gets accumulated with the increasing number of clients.

#Clients	Vogue		[11]	
	Preproc. (sec)	Online (min)	Preproc. (sec)	Online (min)
30	6.97	1.18	7.19	1.29
300	15.65	11.47	18.83	13.04
3000	63.21	115.61	65.71	142.39

TABLE 8: Streaming: Server-side run time for varying number of clients with $K = 16$.

6 Conclusion

Our system Vogue addresses an important problem of identifying heavy hitters in a privacy-preserving manner. It provides secure protocols to identify heavy hitters for two cases: (i) static setting where all the client inputs are available prior to the identification of the heavy hitters, which is computed once, (ii) streaming setting where the inputs arrive in a streaming fashion, and hence heavy hitters have to be computed at frequent intervals of time. Not only does Vogue offer complete privacy in comparison to prior works such as Poplar, but it also improves in terms of providing a faster response time. Our efficiency improvements are attributed to the simplicity of the designed protocols as well as the newly designed compaction protocol, whose round complexity is independent of the size of the input array to be compacted, as opposed to the linear complexity in prior work. We benchmark our system to establish the concrete efficiency improvements of our system, where we witness improvements of up to $10\times$ in run time over Poplar for an increased input length of 512 bits. With respect to the streaming case, the gain is up to $3.5\times$ in comparison to the prior work. Finally, we note Vogue is designed to be modular. Since our protocols make black-box use of various primitives, any efficiency improvements in the same will directly translate to improvements in our protocols for identifying heavy hitters. Moreover, our protocols are generic and can be instantiated with any underlying MPC to obtain the desired level of security.

References

[1] P. Jangir, N. Koti, V. B. Kukkala, A. Patra, B. R. Gopal, and S. Sangal, "Poster: Vogue: Faster computation of private heavy hitters," in *CCS*, 2022.

[2] Y. Zhang, S. Singh, S. Sen, N. G. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *IMC '04*, 2004.

[3] A. Differential Privacy Team, "Learning with privacy at scale," https://machinelearning.apple.com/research/learning-with-privacy-at-scale, 2017, accessed: 2022-09-30.

[4] A. Davidson, P. Snyder, E. Quirk, J. Genereux, and B. Livshits, "Star: Distributed secret sharing for private threshold aggregation reporting," *arXiv preprint arXiv:2109.10074*, 2021.

[5] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Lightweight techniques for private heavy hitters," in *IEEE S&P*, 2021.

[6] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li, "Federated heavy hitters discovery with differential privacy," in *International Conference on Artificial Intelligence and Statistics*, 2020.

[7] R. Bassily, K. Nissim, U. Stemmer, and A. Guha Thakurta, "Practical locally private heavy hitters," *NeurIPS*, 2017.
[8] H. Wu and A. Wirth, "Asymptotically optimal locally private heavy hitters via parameterized sketches," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 7766–7798.
[9] L. Melis, G. Danezis, and E. D. Cristofaro, "Efficient private statistics with succinct sketches," in *NDSS*, 2016.
[10] M. Naor, B. Pinkas, and E. Ronen, "How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior," in *ACM CCS*, 2019.
[11] J. Böhler and F. Kerschbaum, "Secure multi-party computation of differentially private heavy hitters," in *ACM CCS*, 2021.
[12] B. H. Falk and R. Ostrovsky, "Secure merge with o (n log log n) secure operations," in *ITC 2021*, 2021.
[13] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *ACM CCS*, 2021.
[14] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and robust privacy-preserving machine learning," in *USENIX Security*, 2021.
[15] W.-K. Lin, E. Shi, and T. Xie, "Can we overcome the n log n barrier for oblivious sorting?" in *ACM-SIAM Symposium on Discrete Algorithms*, 2019.
[16] S. Dittmer and R. Ostrovsky, "Oblivious tight compaction in o (n) time with smaller constant," in *International Conference on Security and Cryptography for Networks*, 2020.
[17] G. Asharov, I. Komargodski, W.-K. Lin, E. Peserico, and E. Shi, "Oblivious parallel tight compaction," in *Conference on Information-Theoretic Cryptography*, 2020.
[18] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "Optorama: Optimal oblivious ram," in *EuroCrypt*, 2020.
[19] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *ACM CCS*, 2018.
[20] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *ACM CCSW@CCS*, 2019.
[21] A. Patra and A. Suresh, "BLAZE: blazing fast privacy-preserving machine learning," in *NDSS*, 2020.
[22] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *ACM CCS*, 2019.
[23] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.
[24] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, "An efficient secure three-party sorting protocol with an honest majority," *Cryptology ePrint Archive*, 2019.
[25] A. Hasidim, H. Kaplan, Y. Mansour, Y. Matias, and U. Stemmer, "Adversarially robust streaming algorithms via differential privacy," *Advances in Neural Information Processing Systems*, vol. 33, pp. 147–158, 2020.
[26] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren, "Heavy hitter estimation over set-valued data with local differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 192–203.
[27] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.
[28] D. Anderson, P. Bevan, K. Lang, E. Liberty, L. Rhodes, and J. Thaler, "A high-performance algorithm for identifying frequent items in data streams," in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 268–282.
[29] R. J. Wilson, C. Y. Zhang, W. Lam, D. Desfontaines, D. Simmons-Marengo, and B. Gipson, "Differentially private sql with bounded user contribution," *arXiv preprint arXiv:1909.01917*, 2019.
[30] L. Braun, R. Cammarota, and T. Schneider, "A generic hybrid 2PC framework with application to private inference of unmodified neural networks (extended abstract)," in *NeurIPS 2021 Workshop Privacy in Machine Learning*, 2021.
[31] A. Suresh, "Mpcleague: Robust mpc platform for privacy-preserving machine learning," *arXiv preprint arXiv:2112.13338*, 2021.

APPENDIX A
SECURITY OF OUR PROTOCOLS

Lemma A.1. *The protocol, Π_{PHH} (Fig. 10) securely realizes the functionality $\mathcal{F}_{\mathsf{PHH}}$ (Fig. 7) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Compact}})$-hybrid model when instantiated with the MPC of [14].*

Proof sketch Let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}$ denote the corresponding ideal-world adversary. During the emulation of $\mathcal{F}_{\mathsf{Setup}}$, $\mathcal{S}$ establishes common keys with $\mathcal{A}$. These keys are used to sample the common randomness between $\mathcal{S}$ and $\mathcal{A}$ as required throughout the protocol execution. In this way, $\mathcal{S}$ is aware of all the randomness used by $\mathcal{A}$, as well as the shares of the input that $\mathcal{A}$ holds. Following this, $\mathcal{S}$ simulates the steps of the PHH protocol via the simulators of the underlying primitives for equality, comparison, oblivious select, as provided by the MPC of SWIFT [14], [31]. Further, simulation for sorting, shuffling and compaction are performed by $\mathcal{S}$ by emulating the ideal functionalities $\mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Compact}}$, respectively. At the end of the interaction with $\mathcal{A}$, note that $\mathcal{S}$ is aware of the $[\![\cdot]\!]$-shares of $\mathbf{a}$ and $\mathbf{h}$ that $\mathcal{A}$ holds owing to the fact that $\mathcal{S}$ has emulated $\mathcal{F}_{\mathsf{Setup}}$ and simulation of the other primitives happen honestly. Thus, $\mathcal{S}$ now invokes $\mathcal{F}_{\mathsf{PHH}}$ with the $[\![\cdot]\!]$-shares of the output that $\mathcal{A}$ possess to complete the simulation. In summary, the indistinguishability of the simulation of our protocol follows from the fact that the underlying primitives are secure, and $\mathcal{A}$ cannot distinguish between the real-world and ideal-world executions.

Lemma A.2. *The protocol, Π_{Compact} (Fig. 14) securely realizes the functionality $\mathcal{F}_{\mathsf{Compact}}$ (Fig. 11) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Shuffle}})$-hybrid model when instantiated with the MPC of [14].*

Proof sketch As before, let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}$ denote the corresponding ideal-world adversary. The simulation proceeds analogously to as done in the case of simulation for Π_{PHH}. Additionally, here, after invoking $\mathcal{F}_{\mathsf{Shuffle}}$, $\mathcal{S}$ is required to reconstruct the elements in **label**. Recall that **label** is an $\mathcal{N}$-sized vector with unique elements between 1 to $\mathcal{N}$. Since reconstruction of elements in **label** is performed after invoking $\mathcal{F}_{\mathsf{Shuffle}}$, the simulator $\mathcal{S}$ provides shares to $\mathcal{A}$ on behalf of the honest parties, which results in reconstructing **label** such that it is a random sequence of elements between 1 to $\mathcal{N}$.

Similar to the case of Π_{PHH}, the indistinguishability of the simulation for Π_{Compact} follows from the fact that the underlying primitives are secure, and $\mathcal{A}$ cannot distinguish between the real-world and ideal-world executions. Additionally, note that the reconstructed output of shuffle when applied on **label** is always a random sequence of elements between 1 to $\mathcal{N}$ in the real world as well as the ideal world. Hence, $\mathcal{A}$ does not learn any additional information from the reconstructed **label**, and indistinguishability of the simulation holds.

Lemma A.3. *The protocol, Π_{SPHH} (Fig. 17) securely realizes the functionality $\mathcal{F}_{\mathsf{SPHH}}$ (Fig. 15) against a malicious adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Compact}})$-hybrid model when instantiated with the MPC of [14].*

Proof sketch Since Π_{SPHH} does not have any intermediate reconstructions and each step of the protocol invokes secure protocols for the underlying primitives, security follows from the security of the underlying protocols.