Computer Networks 54 (2010) 3309–3326

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks

Tian Bu [a], Jin Cao [a], Aiyou Chen [a], Patrick P.C. Lee [b,*]

[a] Bell Labs, Alcatel-Lucent, 600–700 Mountain Avenue, Murray Hill NJ 07974, USA
[b] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shartin, N.T., Hong Kong

ARTICLE INFO

Article history:
Received 3 May 2009
Received in revised form 27 June 2010
Accepted 28 June 2010
Available online 6 July 2010
Responsible Editor: J.C. de Oliveira

Keywords:
Heavy hitter/changer detection
Network monitoring

ABSTRACT

Identification of significant patterns in network traffic, such as IPs or flows that contribute large volume (heavy hitters) or those that introduce large changes of volume (heavy changers), has many applications in accounting and network anomaly detection. As network speed and the number of flows grow rapidly, identifying heavy hitters/changers by tracking per-IP or per-flow statistics becomes infeasible due to both the computational overhead and memory requirements. In this paper, we propose *SeqHash*, a novel sequential hashing scheme that supports fast and accurate recovery of heavy hitters/changers, while requiring memory just slightly higher than the theoretical lower bound. SeqHash monitors data traffic using a sketch data structure that can flexibly trade-off between the memory usage and the computational overhead in a large range that can be utilized by different computer architectures for optimizing the overall performance. In addition, we propose statistically efficient algorithms for estimating the values of heavy hitters/changers. Using both mathematical analysis and experimental studies of Internet traces, we demonstrate that SeqHash can achieve the same accuracy as the existing methods do but using much less memory and computational overhead.

1. Introduction

Monitoring and detecting significant patterns in a network, such as: (i) the presence of persistent flows with large data volume or (ii) a sudden increase in network traffic due to the emergence of new flows, are essential for network provisioning, management and security because significant patterns often imply events of interests. For instance, a flow that accounts for more than 1% of total traffic may suggest the violation of a service agreement. Similarly, a sudden increase of traffic volume for a specific destination IP may indicate either a hot spot, the beginning of a denial-of-service attack, or traffic rerouting due to link failures elsewhere.

To understand the behavior of a network, we are interested in monitoring a collection of *keys* within a stream of data traffic, where a key is a unique identifier of a flow element. Examples of the definition of a key can be a source IP, a pair of source–destination of IPs, or any combination of the five tuples (i.e., source/destination IPs, source/destination ports, and the protocol). We focus on the identification of two important types of significant patterns: *heavy hitters* and *heavy changers*. A heavy hitter is a key whose traffic volume exceeds a pre-defined threshold, whereas a heavy changer is a key whose change in traffic volume between two monitoring intervals exceeds a pre-defined threshold.[1] Moreover, in order to understand the impact of different heavy hitters and heavy changers, we are also

* Corresponding author. Tel.: +852 31634260.
E-mail addresses: tbu@research.bell-labs.com (T. Bu), cao@research.bell-labs.com (J. Cao), aychen@research.bell-labs.com (A. Chen), pclee@cse.cuhk.edu.hk (P.P.C. Lee).

[1] There are more sophisticated definitions of "change" that account for traffic forecast models. However, the technique we develop in this paper would also apply to such definitions with linear forecast models. Using the simple definition of change allows us to explain our technique more clearly.

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.06.018

interested in finding the *value* associated with each of the heavy hitters and heavy changers, corresponding to the traffic volume and the change in traffic volume, respectively.

However, as the Internet continues to grow in size and complexity, the ever-increasing network bandwidth poses great challenges on monitoring heavy hitters and heavy changers in real time due to its computation and storage requirements. In particular, to identify any network flow that causes significant volume change, the system should scale up to 2^{104} keys.[2] Some fundamental requirements for monitoring and detecting significant patterns in real time for high bandwidth links are discussed below:

- *Fast per-packet update:* The per-packet update speed has to be able to catch up with the link bandwidth even in the worst case when all packets are of the smallest possible size. Otherwise the real time constraint is violated.
- *Fast discovery of significant patterns:* The detection delay of significant patterns should be short such that important events like network attacks and link failures can be responded in time before any serious damage is made.
- *High accuracy:* Both false positive and false negative rates should be minimized. It is well understood that having a false negative may miss an important event and thus delay the necessary reaction. Having a false positive, on the other hand, may trigger unnecessary responses that waste resources.

To monitor data traffic with low memory usage, we propose to keep track of a data stream using a *sketch*, a hash-table-based data structure that provides data stream summaries. In particular, it is desirable to have a sketch that is *reversible*, i.e., by using only the summary information stored in the sketch, we can recover all significant patterns with high accuracy. Having a reversible sketch can facilitate various applications. For instance, if we parallelize network monitoring (e.g., [16]), a subset of threads can be dedicated for recording data packets into sketches, while other threads can periodically collect and aggregate data stream summaries for post processing without requiring the original data traffic as input.

In this paper, we propose effective mechanisms that address the *heavy key detection problem* that is composed of two parts: (i) to identify heavy keys (i.e., either heavy hitters or heavy changers) and (ii) to estimate their associated values. Our main objective is to improve the accuracy of heavy key detection while minimizing both memory usage and computational overhead. Our contributions are summarized as follows:

- We derive a lower bound of memory usage when applying parallel hash tables for heavy key detection at a given error rate.
- We propose *SeqHash*, a sequential hashing scheme that constructs *reversible* multi-level hash arrays for fast and accurate detection of heavy keys while incurring a small increase in memory usage with respect to the derived lower bound. Moreover, we demonstrate via mathematical analysis that SeqHash can flexibly trade off between computational overhead and memory usage using tunable parameters. This allows SeqHash to adapt to different hardware architectures to maximize the overall system performance.
- We design efficient yet accurate methods for estimating the values of heavy keys that we recover from our sketch data structure, by taking into account the information of the counter values and the traffic behavior. Our estimation methods can further reduce errors introduced in the detection stage. This implies that we can make SeqHash more memory efficient by allowing a high error rate in the detection stage and then eliminating the errors in the estimation stage.
- Through extensive experiments using real Internet traces collected at a high speed link, we show that SeqHash yields more accurate results, yet is more memory and computationally efficient, than existing work.

The balance of the paper is organized as follows. In Section 2, we review related work on state-of-the-art data monitoring algorithms. In Section 3, we derive a lower bound of memory requirement when using parallel hash tables for heavy key detection. In Section 4, we describe SeqHash. In Section 5, we formally derive the costs of memory usage and computational overhead of our proposed scheme. In Section 6, we present efficient algorithms for estimating the values associated with all heavy keys. Section 7 shows the evaluation results using Internet traces. Finally, we conclude the paper in Section 8.

2. Related work

To minimize the memory usage of monitoring data traffic, previous studies propose efficient sketch data structures for summarizing data stream information. Examples include heavy hitter detection (e.g., [4–6,8,13]), heavy changer detection (e.g., [9]), or traffic-volume query (e.g., [10]). In particular, given the difficulty of keeping track of all possible per-flow states, Estan and Varghese [6] propose to use a sketch, a set of parallel hash tables to identify heavy hitters. In [6], each packet of a flow is hashed into buckets in different hash tables in parallel using independent hash functions, and the size of the packet is added to the counter associated with each bucket. If all the buckets corresponding to a flow have counter values larger than a predefined threshold, then the flow is known to be a heavy hitter. Similarly, [9] uses a sketch to identify heavy changers. Given a key as input, [6,9] can tell, with high accuracy, whether the input key is heavy. However, the sketches used in [6,9] are *irreversible*, meaning that it is computationally infeasible to recover *all* heavy keys using only the sketch-based summaries (see Section 4 for details).

Deltoids [7] and reversible sketch [17] are two approaches that use reversible sketch data structures to ad-

[2] This number is calculated based on the number of possible five-tuple flows: source IP address (32 bits), source port (16 bits), destination IP address (32 bits), destination port (16 bits), and protocol (8 bits). The number may be significantly smaller for realistic traffic since not all possible combination of these fields are possible.

dress both heavy hitter detection and heavy changer detection. In Deltoids [7], the key space is partitioned into different groups using independent 2-universal hash functions. Each group contains L counters, where L is the bit length of a key and each counter corresponds to a particular bit within a key. If each group contains exactly one heavy key (either heavy hitter or heavy changer) based on a group test, then the heavy key will be recovered. Otherwise, the whole group will be discarded, even though it may review partial information (e.g., a subset of bits) regarding a heavy key. As a result, more buckets, and hence more memory, are required in order to recover all heavy keys with high accuracy.

Reversible sketch [17] proposes a modular hashing scheme to narrow down a candidate set of heavy keys. In particular, a key is divided into multiple sub-keys with smaller bit lengths, where the candidate sub-keys are then combined to construct the actual heavy keys. However, as we will explain in Section 4.4, reversible sketch incurs a higher collision probability of hashing a non-heavy key and a heavy key into the same bucket. Also, it is computationally expensive in recovering all heavy keys, as its complexity is sub-linear of the size of the key space [17].

By carefully engineering a different sketch data structure, SeqHash improves the approaches of Deltoids and reversible sketch in heavy key detection in terms of both computational overhead and memory usage. Also, SeqHash provides tunable parameters that can trade-off between the computational overhead and memory usage. We present the analysis in subsequent sections.

3. Memory lower bound for a hash array

In this section, we derive the lower bound of the memory required for identifying heavy keys using parallel hash tables as in [6]. We also present the trade-off analysis between the memory usage and computational overhead. This section aims to lay out a baseline understanding of the memory-computation trade-off analysis for SeqHash that we propose later.

Table 1 summarizes the major notation used in this paper. We write "log" to denote the natural logarithm, and "$\log_2$" to denote the logarithm with base 2.

We model network traffic within a monitoring interval as a stream of data packets that arrive in order, where each packet is represented by a pair (x, v_x), where x denotes a key in the key space of size N, and v_x denotes the value associated with key x. Throughout this paper, we set v_x to be the size of a packet for key x. Clearly, the identification of heavy keys (i.e., either heavy hitters or heavy changers) is straightforward if the value of v_x for each possible key x is known, that is, we associate a counter with each possible key. However, keeping track of every possible pair (x, v_x) becomes infeasible for large N, as we require a total of N counters.

To identify heavy keys with a limited amount of memory, we construct a *hash array* (or *sketch*) to approximate the set of heavy keys. Fig. 1 illustrates the structure of a hash array, which consists of M hash tables with K buckets each. Each given key is hashed to a bucket within each hash table using an independent hash function. The value

Table 1
Major notation used in the paper.

Defined in Section 3	
x	Key
v_x	Value associated with key x
t	Pre-specified threshold to identify heavy keys
N	Size of key space
M	Total number of hash tables
K	Size of a hash table (in number of buckets)
U	Size of memory in total number of buckets, i.e., $M \times K$
H	Maximum number of heavy keys being observed
$\mathcal{C}$	Candidate set of heavy keys
ϵ	False positive rate (Eq. (2))
Defined in Section 4	
D	Number of words in a key (= number of hash arrays)
M_i	Number of hash tables in hash array i, where $1 \leqslant i \leqslant D$
$\mathcal{T}_{i,j}$	The jth hash table in hash array i, where $1 \leqslant i \leqslant D$, $1 \leqslant j \leqslant M_i$
$f_{i,j}$	$\{0,\ldots,N_i - 1\} \to \{1,\ldots,K\}$, the hash function for table $\mathcal{T}_{i,j}$
w_i	The ith word of a key, where $1 \leqslant i \leqslant D$
b_i	Number of bits in word w_i, where $1 \leqslant i \leqslant D$
$\mathcal{N}_i$	Sub-key space, where $1 \leqslant i \leqslant D$
N_i	$2^{b_1+\cdots+b_i}$, size of sub-key space $\mathcal{N}_i$, where $1 \leqslant i \leqslant D$
$\mathcal{H}_i$	Set of sub-keys of the heavy keys in original key space
$\mathcal{C}_i$	Intermediate candidate set of sub-keys of the heavy keys
Defined in Section 5	
α	Intermediate false positive rate

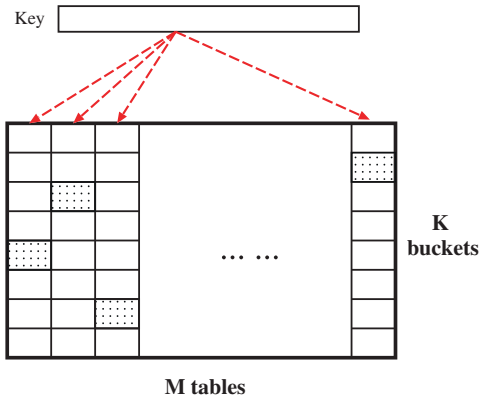


Fig. 1. A hash array, which consists of M hash tables with K buckets each.

v_x is then added to the counter associated with the bucket. The hash function for each table is chosen independently from a class of hash functions such that the K buckets of each table form a random partition of N keys.

Here, we let H be the maximum number of heavy keys that can be observed over a monitoring period, and we provision memory based on H. Throughout the paper, we assume that H is also the *actual* number of heavy keys, and this enables us to conduct the worst-case analysis.

We first derive the memory lower bound for heavy hitter detection, for which the hash array proposed by [6] is used. We then discuss how we apply the results to heavy changer detection as well.

3.1. Memory lower bound for heavy hitter detection

Recall that a heavy hitter refers to a key x whose sum of values $\sum v_x$ exceeds a pre-specified threshold t. We call a

bucket to be *heavy* if its counter value (i.e., sum of values of all keys hashed to the bucket) crosses the threshold t. It is easy to see that for any heavy hitter, every bucket that it falls into in each of the M tables is a heavy bucket. A candidate set $\mathcal{C}$ of heavy hitters refers to the set of keys whose buckets within the M hash tables are all heavy buckets. Note that $\mathcal{C}$ is the superset of the actual heavy hitters, and it may contain some non-heavy hitters that happen to fall into heavy buckets in all M hash tables.

Our lower-bound analysis assumes that the traffic distribution is very skewed such that the sum of values of any subset of non-heavy hitters is less than the threshold, i.e., the contributions of non-heavy hitters are negligible. If the non-heavy hitters are non-negligible, then it is possible that a heavy bucket is contributed by only the non-heavy keys whose sum of values exceeds the pre-specified threshold, and hence the false positive rate increases. As a result, we need more memory (or buckets) for a hash array to satisfy a false positive rate. Nevertheless, our memory lower-bound remains valid, although it denotes a loose lower bound when the "noise" values due to non-heavy keys become significant.

Lemma 1. *Let Z be the number of heavy hitters contained in an arbitrary bucket. Then Z follows a binomial distribution with parameters H and $\frac{1}{K}$. If H is large (e.g., $H > 100$), then Z can be approximated by a Poisson distribution with parameter H/K.*

Remark. The proof is straightforward and is omitted. When $H/K = \log 2$ (see Theorem 1 below), Lemma 1 indicates that about 50% of the buckets within a hash table do not contain any heavy hitters, and that among the heavy buckets within a hash table, about 70% of them contain exactly one heavy hitter.

Lemma 2. *The expected size of the candidate set $\mathcal{C}$ of heavy hitters is given by $E|\mathcal{C}| \approx H + (N-H)\left(1-\left(1-\frac{1}{K}\right)^H\right)^M$. When H is large, then*

$$E|\mathcal{C}| \approx H + (N-H)(1-e^{-H/K})^M. \quad (1)$$

Proof. Let p_e be the probability that a non-heavy hitter falls into the candidate set $\mathcal{C}$ of heavy hitters, and p_i be the probability that a non-heavy hitter falls into a heavy bucket in the ith table, where $1 \leqslant i \leqslant M$. If a key is hashed to each bucket uniformly and independently, then the probability that a bucket contains no heavy hitter is given by $\left(1-\frac{1}{K}\right)^H$. Hence, $p_i = 1-\left(1-\frac{1}{K}\right)^H$. As hash functions that we select are assumed to be independent, we have $p_e = \prod_{i=1}^{M} p_i$. Hence, we have $E|\mathcal{C}| = H + (N-H)p_e$, and by Lemma 1, the result follows. □

For the set $\mathcal{C}$, let ϵ be the *false positive rate*, defined as the ratio of the expected number of false positives to H, i.e.,

$$E|\mathcal{C}| = H + \epsilon H. \quad (2)$$

Then by (1), for a given value ϵ and a large H, the number of tables within a hash array is

$$M \approx -\frac{\log(N\epsilon^{-1}H^{-1})}{\log(1-e^{-H/K})}. \quad (3)$$

Let $U = M \times K$ denote the memory size, in total number of buckets, of a single hash array. Two questions naturally arise: (i) Given a fixed false positive rate ϵ, what is the minimum memory size U? (ii) Given a fixed memory size U, what is the minimum false positive rate ϵ? The following two theorems state the minimum amount of memory required for a specified false positive error ϵ.

Theorem 1. *Given a fixed false positive rate ϵ, the memory size U is minimized when $K = H/\log 2$ and $M = \log_2\left(\frac{N}{\epsilon H}\right)$. Also, the minimum value of U is $\frac{H}{\log 2}\log_2\left(\frac{N}{\epsilon H}\right)$.*

Proof (*Sketch*). Note that U is a function of K, such that $U = -\frac{K\log(N\epsilon^{-1}H^{-1})}{\log(1-e^{-H/K})}$, where ϵ, N, and H are fixed parameters. By differentiating U with respect to K and setting the derivative $\frac{dU}{dK} = 0$, we can see that U is minimized when $K = H/\log 2$, and by (3), $M = \log_2\left(\frac{N}{\epsilon H}\right)$. □

Remark. Theorem 1 is essentially the same memory optimization problem in the design of Bloom filter [1], whose network-related applications are surveyed in [2].

Theorem 1 specifies the corresponding number of hash operations M when U is minimized. However, it is important to note that we can observe a trade-off between the memory requirement and the hash operations in order to achieve a fixed false positive error ϵ. Fig. 2 shows the trade-off between M and U for the case where $N = 2^{32}$, $H = 1000$ in the lower bound case. The circles represent the optimal pair of (M,U) such that U is minimized. To achieve the same expected normalized false positive error ($\epsilon = 10^{-6}$ or $\epsilon = 10^{-3}$), we can in fact use just half of the optimal number of hashing tables with the price of increasing the memory size by about only 20%. This may be desirable when hash operations are considered expensive.

Theorem 2. *Given a fixed memory size U, the false positive rate ϵ is minimized when $K = H/\log 2$.*

Proof. We again can express ϵ as a function of K for a given U, N, and H. Instead of differentiating ϵ with respect to K, we provide a more intuitive proof for the theorem. Suppose

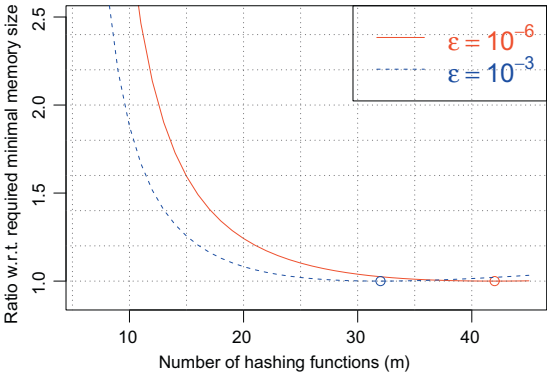


Fig. 2. The trade-off between the total memory and the number of hashing operations. The y-axis is the normalized memory size with respect to the minimum memory size for a given ϵ.

the contrary that ϵ is minimized at $\epsilon = \epsilon^*$ when $K = K^*$, where $K^* \neq H/\log 2$, for a fixed $U = U^*$. By Theorem 1, we can find a smaller U' at $K = H/\log 2$, where $U' < U^*$, that satisfies the error rate ϵ^*. Since U is a strictly decreasing function of ϵ, then there exists a smaller $\epsilon < \epsilon^*$ that satisfies U^*. This contradicts the fact that ϵ^* is minimum for $U = U^*$. Therefore, ϵ is minimized when $K = H/\log 2$. □

3.2. Memory lower bound for heavy changer detection

We provide high-level arguments to justify for a given false positive rate, the memory size required for heavy changer detection is no less than that for heavy hitter detection. For a given bucket, let $y^{(1)}$ and $y^{(2)}$ be the bucket values in the previous and current monitoring interval, respectively, and let $y = y^{(2)} - y^{(1)}$ be the change in the bucket value across the two monitoring intervals. For the heavy changer case, a bucket is considered *heavy* if $|y|$ crosses a pre-specified threshold t. Let us assume that the change of value of every non-heavy changer is non-negligible. However, unlike the heavy hitter case presented above, it is now possible that a positive heavy changer (i.e., with change of value greater than t) and a negative heavy changer (i.e., with change of value less than $-t$) are hashed to the same bucket such that the bucket is *not heavy* (i.e., $|y| < t$). Therefore, the outcome of the threshold test may not include all heavy changers, and there will be false negatives in addition to false positives when using the intersections of heavy buckets to identify the heavy changers. To minimize the false negative errors, [17] introduces a notion of *misses*, such that a key is still considered to be a heavy key if it falls into at least $M - r$ heavy buckets, where r is the number of allowed misses. The use of misses has a trade-off of increasing the false positive rate. Hence, we need more memory (buckets) to satisfy a given false positive rate. Note that the memory lower bound derived for the heavy hitter case remains valid for the heavy changer case, though it denotes a looser lower bound for the latter.

4. SeqHash

In this section, we present SeqHash, a sequential hashing scheme that enables us to recover all heavy keys (either heavy hitters or heavy changers) given the counter values stored in a reversible sketch data structure.

4.1. Motivation

Suppose that we use a single hash array (Fig. 1 in Section 3) to keep track of the heavy keys. One major limitation of such a data structure is that it is *irreversible*, meaning that it is computationally infeasible to recover all heavy keys given the counter values stored in the hash array. This irreversibility holds even if the non-heavy keys have negligible values. To understand why this limitation exists, note that the hash array corresponds to many-to-one mappings from the entire key space of size N to a small number of buckets. To identify all heavy keys from the hash array, the only solution is to exhaustively enumerate all possible keys in the entire key space, and check if each individual key is associated with a heavy key in each hash table. Such an approach, however, is computationally infeasible if N, the size of the key space, is very large.

In view of this, we propose SeqHash, a framework of using *multi-level hashing* to recover H heavy keys from a very large key space. The multi-level hashing scheme allows us to divide the original problem into much smaller sub-problems to which exhaustive search can be applied.

4.2. Intuition of SeqHash

To illustrate the general idea of SeqHash, for a key x with $n = \log_2 N$ bits, we first focus on identifying a sub-key of x with b bits that belongs to a heavy key. We assume b is sufficiently small (say 4–8 bits) such that enumeration of this sub-key space for identifying the heavy sub-keys is *trivial* using a hash array as described in Section 3. Next, we combine the heavy sub-keys that have just been found with additional bits (say 2–4 bits) of the key to form a larger sub-key with more bits, say $b' > b$ bits. Enumeration of this larger sub-key space (with b' bits) is now significantly reduced because the smaller sub-keys (with b bits) for heavy keys are already known. Therefore, we can again use a new hash array to identify the larger sub-keys of the heavy

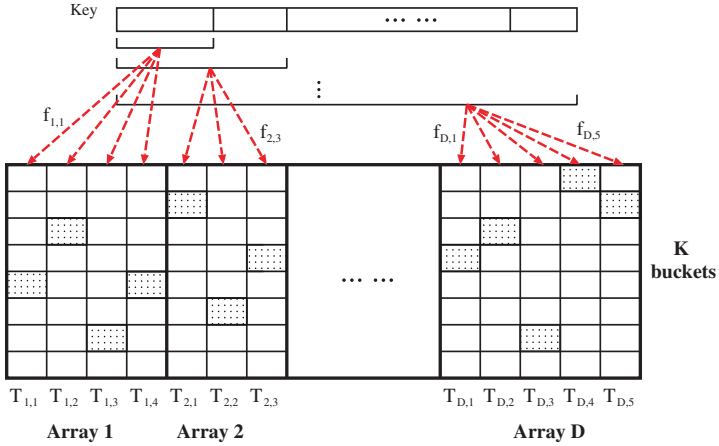


Fig. 3. Overview of SeqHash: how a key is mapped to multiple hash arrays.

keys. Repeating the process, we can eventually discover the full heavy keys in the original key space.

4.3. Design of SeqHash

SeqHash uses a sketch-based data structure that is composed of multiple hash arrays. Fig. 3 depicts the relationship between a key and the multiple hash arrays in SeqHash, and additional notation is presented in Table 1. We partition a key x into D words $w_1 w_2, \ldots, w_D$ such that each word w_i has b_i bits, where $1 \leqslant i \leqslant D$. Let us consider the sub-key $w_1, \ldots, w_i$, formed by the first i words of key x. The sub-key $w_1, \ldots, w_i$ is a member of the sub-key space $\mathcal{N}_i = \{0, 1, \ldots, N_i - 1\}$, where $N_i = 2^{\sum_{r=1}^{i} b_r}$. In each sub-key space $\mathcal{N}_i$, we let $\mathcal{H}_i$ denote the set of sub-keys that correspond to the heavy keys in the original key space. We can easily tell $\mathcal{H}_i$ is at most of size H. Also, our data structure consists of D hash arrays, in which the ith hash array corresponds to sub-key $w_1, \ldots, w_i$ and contains M_i hash tables $\mathcal{T}_{i,1}, \ldots, \mathcal{T}_{i,M_i}$. The total number of hash tables is given by $M = \sum_{i=1}^{D} M_i$.

SeqHash consists of two major steps: (1) *Update* step, which adds the value of a key to the associated buckets of the hash arrays, and (2) *Detection* step, which finds the set of heavy keys. The Update step is carried out for every incoming key (i.e., data packet), while the Detection step is carried out at the end of every monitoring interval.

Algorithm 1 outlines the Update step. For each incoming key $x = w_1, \ldots, w_D$ with value v_x, we associate the sub-key $w_1, \ldots, w_i$ with bucket $f_{i,j}(w_1, \ldots, w_i) \in \{1, \ldots, K\}$ in hash table $\mathcal{T}_{i,j}$, where $1 \leqslant i \leqslant D$, $1 \leqslant j \leqslant M_i$, and $1 \leqslant k \leqslant K$. We then increment the counter in the bucket with value v_x.

Algorithm 1: Update step

Input: a key x with value v_x
1: Partition key x into D words as $w_1 w_2, \ldots, w_D$, where word w_i has b_i bits for $1 \leqslant i \leqslant D$
2: **for** i = 1 to D **do**
3: **for** j = 1 to M_i **do**
4: Increment the counter of bucket $f_{i,j}(w_1, \ldots, w_i)$ in hash table $\mathcal{T}_{i,j}$ with value v_x

Algorithm 2: Detection step

Inputs: hash tables $\{\mathcal{T}_{i,j}\}_{1 \leqslant i \leqslant D, 1 \leqslant j \leqslant M_i}$ with heavy buckets
Output: a set of heavy keys
1: Set $\mathcal{C}_0 = \{0\}$ and $\mathcal{C}_i = \phi$ for $1 \leqslant i \leqslant D$
2: **for** i = 1 to D **do**
3: **for all** $x' \in \mathcal{C}_{i-1}$ **do**
4: **for** w_i = 0 to $2^{b_i} - 1$ **do**
5: $x'' = x' \times 2^{b_i} + w_i$
6: **if** *is_heavy*(x'', i) == TRUE **then**
7: Add x'' to $\mathcal{C}_i$
8: return $\mathcal{C}_D$

Algorithm 2 summarizes the Detection step for recovering heavy keys. The main idea is to decompose the original problem of finding H heavy keys into a sequence of D nested sub-problems, each of which determines a candidate set $\mathcal{C}_i$ of sub-keys from subspace $\mathcal{N}_i$ as an approximation of $\mathcal{H}_i$. We first identify $\mathcal{C}_1$ by searching for all values in $\mathcal{N}_1$ that have all their associated buckets in $\mathcal{T}_{1,1}, \ldots, \mathcal{T}_{1,M_1}$ considered to be heavy by the function *is_heavy*, whose actual implementation depends on whether we consider heavy hitters or heavy changers, and is described later. To determine $\mathcal{C}_i$ for $2 \leqslant i \leqslant D$, we first concatenate each sub-key $x' \in \mathcal{C}_{i-1}$ with an arbitrary word $w_i \in \{0, \ldots, 2^{b_i} - 1\}$ to form x''. We then check whether x'' is considered to be heavy in hash array i by the function *is_heavy*. If *is_heavy* returns TRUE, then it means x'' is a candidate sub-key and we include x'' into $\mathcal{C}_i$. We continue this process and finally return $\mathcal{C}_D$, which is our final set $\mathcal{C}$ of candidate heavy keys.

We now present the implementation of the function *is_heavy*. Fig. 4 presents the pseudo-code of *is_heavy* for heavy hitter detection, whose goal is to identify the keys that are *always* associated with heavy buckets. The function returns FALSE if a bucket has its counter value less than some pre-specified threshold t in the current monitoring interval.

The implementation of *is_heavy* is slightly more complicated for heavy changer detection. As explained in Section 3.2, it is possible that positive and negative heavy changers cancel each other if they are hashed to the same bucket, and this leads to false negatives. We borrow the idea of *misses* (i.e., non-heavy buckets) in [17] and extend this idea as follows. Let $y^{(1)}$ and $y^{(2)}$ be the counter values of a given bucket in the previous and current monitoring intervals, respectively. We observe that for a change $|y^{(2)} - y^{(1)}|$ to exceed a pre-specified threshold t, then we must have either $y^{(1)} > t$ or $y^{(2)} > t$ (or both). With this observation, we define a miss to be *legitimate* if a non-heavy bucket has $y^{(1)} > t$ or $y^{(2)} > t$. Our detection only allows the legitimate misses and eliminates any miss that is due to only non-heavy keys.

Fig. 5 illustrates the implementation of *is_heavy* for heavy changer detection. Let r_i be an input parameter that specifies the number of allowed legitimate misses for the ith hash array. Then *is_heavy* returns FALSE if bucket $f_{i,j}(x'')$ is a non-legitimate miss, or the number of legitimate misses for hash array i exceeds r_i.

Note the performance of SeqHash depends on the choices of parameters, and hence the layout of the multi-hash-array structure. We discuss this issue in Section 5.

4.4. Modular hashing

In addition to SeqHash, another possible implementation of multi-level hashing is called *modular hashing*, which is the core component of reversible sketch in [17]. Fig. 6

function *is_heavy*(x'', i)
/* Heavy hitter detection */
1: **for** j = 1 to M_i **do**
2: y = counter value of bucket $f_{i,j}(x'')$ in current interval
3: **if** y less than threshold t **then**
4: **return** FALSE
5: **return** TRUE

Fig. 4. Implementation of *is_heavy* for heavy hitter detection.

shows the idea of modular hashing. Let us consider a hash array of M hash tables with K buckets each. In modular hashing, a key is partitioned into multiple words, each of which is independently hashed to a value with fewer bits. The hash values of all words are then concatenated to form the bucket index in a hash table. To recover heavy keys, we perform a reverse mapping from a bucket index to a word, that is, we examine the index of each heavy bucket and determine the words that are hashed to the index. The set of the recovered heavy keys will be given by the intersections of such words, such that each word of a recovered heavy key is hashed to the corresponding subset of bits of a heavy bucket index. We can apply an iterative approach as in [17] to find the intersections and recover the heavy keys.

However, modular hashing has a couple of design limitations. First, to make the reverse mapping from a bucket index to a word possible, the number of buckets (i.e., K) in each hash table must be in the form of 2^{qc}, where q is the number of partitioned words of a key and c is some integer. This limits the choice of K. Thus, modular hashing is less flexible in trading off between computational overhead and memory usage through tuning parameters when compared to SeqHash (see Section 5).

Second, the correlation structure among the IP addresses will significantly increase the false positive rate.

```
function is_heavy(x'', i)
/* Heavy changer detection */
 1: Set r = 0
 2: for j = 1 to M_i do
 3:    y^(1) = counter value of bucket f_{i,j}(x'') in previous interval
 4:    y^(2) = counter value of bucket f_{i,j}(x'') in current interval
 5:    if |y^(2) − y^(1)| less than threshold t then
 6:       if non-legitimate miss then
 7:          return FALSE
 8:       else
 9:          r = r + 1
10:          if r > r_i then
11:             return FALSE
12: return TRUE
```

Fig. 5. Implementation of *is_heavy* for heavy changer detection.

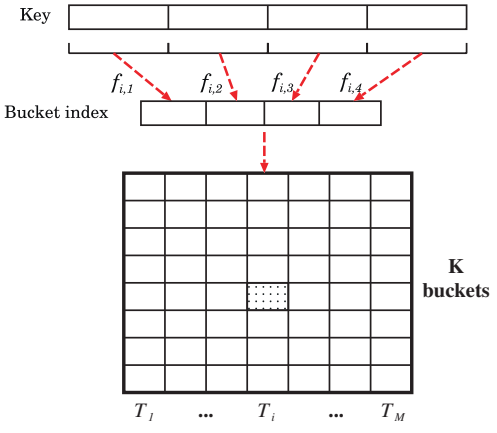


Fig. 6. Overview of modular hashing.

For example, suppose that the key space is the set of 32-bit IP addresses and that we partition the IP addresses into $q = 4$ octets. Let $x_1 \cdot x_2 \cdot x_3 \cdot x_4$ be a heavy key. Then every non-heavy key $x_1 \cdot x_2 \cdot x_3 \cdot ^*$ will have probability $\frac{1}{8}$ to be hashed to a heavy bucket in a hash table, as opposed to $\frac{1}{K}$ in the scheme that hashes the entire key (note that if a 2-universal hash function is used, then any two different keys will be hashed to the same bucket with probability $\frac{1}{K}$). To destroy the correlation structures of keys, [17] proposes to apply a *mangling* function to each of the input keys to generate a randomized key. However, the accuracy of modular hashing depends on the choice of the mangling function. On the other hand, SeqHash does not assume any prior mangling on input keys, as it includes the hash operations that apply to the entire key and possibly remove any correlation of keys. In Section 7, we compare both SeqHash and modular hashing.

5. Analytical evaluation

In this section, we present a formal analysis of SeqHash in terms of memory and computation, and discuss how the choices of parameters achieve the most savings in both memory and computation for a targeted false positive rate. We also conduct complexity comparison between SeqHash, Deltoids, and reversible sketch.

Our analysis assumes that non-heavy keys have negligible contribution to the counter values. In addition, we focus on the heavy hitter case. This enables us to directly compare our results with the memory lower bound we derive in Theorem 1 (see Section 3), which gives a *tight* bound. As explained in Section 4.1, using a single hash array to recover all heavy hitters incurs a computational complexity $\Theta(N)$ (i.e., by enumerating all possible keys in the entire key space), even though all non-heavy hitters have negligible values. We show that with the right design choice, SeqHash can reduce the computational complexity to $\Theta(H\log N)$ with a slight increase in total memory with respect to the memory lower bound we derive in Section 3.

We also discuss how we extend our analytical results to heavy changer detection, under the assumption that non-heavy keys are negligible. To evaluate the case with significant non-heavy keys, we resort to the use of trace-driven experiments, as described in Section 7.

In the following analysis, in addition to assuming negligible non-heavy hitters, we assume that the maximum number of heavy keys H is large (e.g., $H > 100$), and that the size of the sub-key space $N_i \gg H$, for $1 \leqslant i \leqslant D$ (e.g., $N_i \geqslant 64H$). Such assumptions enable us to estimate the number of heavy keys in each intermediate step, as discussed below.

5.1. Memory cost for the update step

SeqHash uses a multi-hash-array data structure (see Fig. 3). In this subsection, we derive the memory (in total number of buckets) required for this data structure given a fixed false positive rate ϵ. Our goal is to compute K (i.e., the number of buckets within each hash table) and M_i

(i.e., the number of hash tables within each hash array i, where $1 \leqslant i \leqslant D$).

First, we determine K. We note that $\mathcal{H}_i$, the set of distinct values in the sub-key $w_1,\ldots,w_i$ of the heavy keys, has the maximum size H. Suppose that we focus on the worst case such that the heavy keys are randomly distributed over the entire key space. Since we assume that $N_i \gg H$, we have $|\mathcal{H}_i| \approx H$ for $1 \leqslant i \leqslant D$. Also, as H is large, by Lemma 1, the minimum memory requirement is fulfilled when $K = H/\log 2$, which is independent of the size of the sub-key space. As a result, we have $K = H/\log 2$.

To derive M_i for $1 \leqslant i \leqslant D$, we need to compute $E|\mathcal{C}_i|$, the expected size of the intermediate candidate set of heavy keys $\mathcal{C}_i$. Note that $\mathcal{C}_i$ is returned by each iteration in the Detection step (see Algorithm 2) and is determined by the concatenation of the previous intermediate candidate set $\mathcal{C}_{i-1}$ and the set $\{0,\ldots,2^{b_i}-1\}$. Thus, using the proof of Lemma 2 (see Section 3) and the facts that $|\mathcal{H}_i| \approx H$ and $K = H/\log 2$, the expected size of $\mathcal{C}_i$ (conditioned on $\mathcal{C}_{i-1}$) is

$$\begin{aligned} E|\mathcal{C}_i| &= |\mathcal{H}_i| + \left(|\mathcal{C}_{i-1}| \times 2^{b_i} - |\mathcal{H}_i|\right) \\ &\quad \times \left(1 - \left(1 - \frac{1}{K}\right)^{|\mathcal{H}_i|}\right)^{M_i} \\ &\approx H + \left(|\mathcal{C}_{i-1}| \times 2^{b_i} - H\right) \times \frac{1}{2^{M_i}}. \end{aligned} \tag{4}$$

Let α be the intermediate false positive rate such that the expected size of $\mathcal{C}_i$ is given by $E|\mathcal{C}_i| = (1+\alpha)H$ for $1 \leqslant i \leqslant D-1$ (note that the sizes of the initial and final candidate sets are $E|\mathcal{C}_0| = 1$ and $E|\mathcal{C}_D| = (1+\epsilon)H$, respectively). Then we have M_i as follows:

$$M_i = \begin{cases} \log_2\left(\frac{2^{b_1}-H}{\alpha H}\right) & \text{if } i = 1, \\ \log_2\left(\frac{2^{b_i}(1+\alpha)-1}{\alpha}\right) & \text{if } 2 \leqslant i \leqslant D-1, \\ \log_2\left(\frac{2^{b_D}(1+\alpha)-1}{\epsilon}\right) & \text{if } i = D. \end{cases} \tag{5}$$

Therefore, the memory of the multi-hash-array structure used by SeqHash is

$$\begin{aligned} \text{Memory} &= K \times \sum_{i=1}^{D} M_i \\ &\leqslant \frac{H}{\log 2} \times \left[\log_2\left(\frac{2^{b_1}}{\alpha H}\right) + \sum_{i=2}^{D-1} \log_2\left(\frac{2^{b_i}(1+\alpha)}{\alpha}\right) \right. \\ &\quad \left. + \log_2\left(\frac{2^{b_D}(1+\alpha)}{\epsilon}\right)\right] \\ &= \frac{H}{\log 2} \times \log_2\left(\frac{N}{\epsilon H} \times \left(1+\frac{1}{\alpha}\right)^{D-1}\right). \end{aligned} \tag{6}$$

In comparison to the result of Theorem 1, SeqHash requires $K(D-1)\log_2\left(1+\frac{1}{\alpha}\right)$ additional buckets, where $K = H/\log 2$, and D and α are fixed parameters that are tunable depending on applications. We evaluate this additional memory overhead, together with the computational cost derived in the next subsection, later in Section 5.3.

It is important to note that the choices of the design parameters are based on the value of H, the maximum number of heavy keys being observed. To estimate H for a pre-specified threshold t, one conservative method is to set $H = Y/t$, where Y is the total data volume observed within a monitoring interval and can be estimated based on the history of traffic distribution. For example, the number of heavy hitters that exceeds 1% of the traffic is at most 100. Note that this approach applies to the detection of both heavy hitters and heavy changers, since the maximum value of every key is upper bounded by Y. Our future work is to derive a more accurate H so as to reduce the memory usage.

5.2. Computational cost for detection

We now evaluate the computational cost, in number of hash operations, of recovering all heavy keys in the Detection step for heavy hitter detection. (see both Algorithm 2 and Fig. 4). Note that for a non-heavy hitter, the function *is_heavy* returns FALSE immediately when it hits a non-heavy bucket whose value is less than the pre-specified threshold, and hence it skips subsequent hash operations. Since with $K = H/\log 2$, about 50% of buckets are non-heavy buckets (see Lemma 1 in Section 3). Therefore, the expected number of hash operations performed on a non-heavy hitter is approximately equal to 2. On the other hand, the number of hash operations performed on a heavy hitter is equal to the total number of hash tables. With this observation, and using the facts that: (i) $|\mathcal{H}_i| \approx H$ for $1 \leqslant i \leqslant D$, (ii) $|\mathcal{C}_0| = 1$, and (iii) $|\mathcal{C}_i| \approx (1+\alpha)H$ for $1 \leqslant i \leqslant D-1$, the computational cost of the Detection step is as follows:

$$\begin{aligned} \text{Computation} &\approx \sum_{i=1}^{D} \left[M_i \times |\mathcal{H}_i| + 2 \times \left(|\mathcal{C}_{i-1}| \times 2^{b_i} - |\mathcal{H}_i|\right)\right] \\ &\leqslant H\log_2\left(\frac{N}{\epsilon H} \times \left(1+\frac{1}{\alpha}\right)^{D-1}\right) + 2 \\ &\quad \times \left[2^{b_1} + (1+\alpha)H\sum_{i=2}^{D} 2^{b_i}\right] (\text{from (6)}). \end{aligned} \tag{7}$$

5.3. Memory-computation trade-off

The memory and computational costs of SeqHash depend on the tunable parameters ϵ, α, and the set of b_i's (which in turn determine D). Suppose that the following parameters are considered: $N = 2^{32}$, $\epsilon = 0.2\%$, $H = 500$ (and hence $K = H/\log 2 \approx 722$), $b_1 = 16$ (and hence $N_1 = 2^{16}$ and $N_i \geqslant 64H$), and $b_i = b$ for $i \geqslant 2$, where we fix $b = 1$, 2, and 4. We then vary α to obtain the corresponding pairs of memory and computational costs using (6) and (7). Fig. 7 illustrates how SeqHash trades off between the memory and computational costs. When $b = 1$ or 2, a smaller detection cost is obtained as compared to $b = 4$, while the difference between $b = 1$ and 2 is very small. For example, when $b = 2$ and $\alpha = 9$, the total number of tables across all hash arrays is given by $M = \sum_{i=1}^{D} M_i \approx 33$ (where $M_1 = 4$, $M_i = 2$ for $2 \leqslant i \leqslant D-1$, $M_D = 15$, and $D = 9$), while the computational cost for detection is about 400 K hash operations

and is about twice the minimum computational cost achieved by the use of large memory. Note that the number of tables in the minimum memory requirement is $\log_2 \frac{N}{\epsilon H} = 32$, where the heavy key detection is done by enumeration of the entire key space. Thus, with only one extra table, we can recover all heavy keys with manageable computational overhead.

5.4. Complexity results

We now derive the complexity results of different measures that quantify the costs of the Update and Detection steps of SeqHash. Such measures are also considered by [17] to evaluate the Reversible Sketch scheme.

The complexity of SeqHash depends on the choices of parameters. Here, we focus on a particular set of design parameters where $\alpha = 9$, $b_1 = 16$, and $b_i = 2$ for $i \geqslant 2$. This also implies $D = (\log_2 N - 16)/2 + 1 = \Theta(\log N)$. As justified by the results of Fig. 7 and our experiments (see Section 7), this set of parameters provides manageable memory and computational costs. In addition, we set $\epsilon H = \Theta(1)$, meaning that the number of false positives is controlled within a constant factor. This setting enables us to reduce our complexity results to functions of N and H for comparison with existing approaches.

Memory used by the update step: The total memory is given by the total number of buckets within the multi-hash-array structure, and the result is shown in (6). Note that $\log_2(1+\frac{1}{\alpha})^{D-1} = (D-1)\log_2(1.1)$ is smaller than $\log_2 N$ in general. Thus, the complexity is given by $\Theta(H \log N)$.

Number of memory accesses of the update step: The number of memory accesses is equal to the number of tables within the multi-hash array structure, i.e., $\Theta(\log N)$.
Number of hash operations of the update step: This is equal to the number of memory accesses, as each hash operation corresponds to a unique hash table.

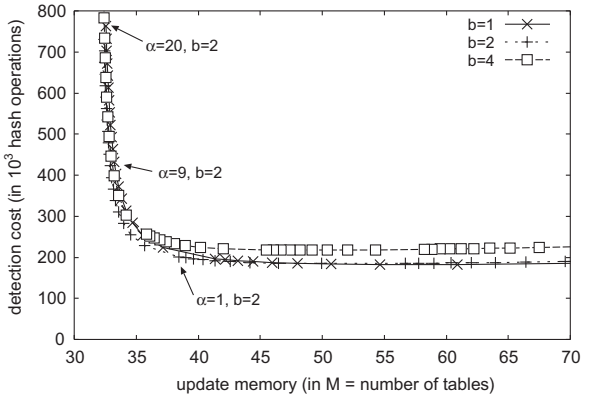


Fig. 7. The memory-computation trade-off of SeqHash, with $N = 2^{32}$ and $H = 500$. Note that the x-axis represents the total number of hash tables across the multi-hash-array structure, and the y-axis represents the number of hash operations of the detection step (for heavy hitter detection). We highlight the memory and computational costs for different values of α when $b = 2$.

Memory used by the detection step: In addition to the multiple hash arrays, SeqHash also stores $(1+\alpha)H = 10H = \Theta(H)$ of the candidate heavy keys in intermediate steps (see Algorithm 2 in Section 4). Overall, the complexity is given by $\Theta(H \log N)$.
Number of hash operations of the detection step: The result is shown in (7). Note that $2^{b_1} + (1+\alpha)H\sum_{i=2}^{D} 2^{b_i} = \Theta(H \log N)$. Thus, the complexity is given by $\Theta(H \log N)$.

Table 2 compares the complexity results of SeqHash with Deltoids [7] and reversible sketch [17]. Note that although the complexity results of Deltoids and reversible sketch in Table 2 are derived for heavy changer detection, they remain valid for heavy hitter detection, which we assume in our complexity analysis. Compared with Deltoids, SeqHash has the same complexity results. However, as shown in our experiments (see Section 7), SeqHash has significantly higher accuracy than Deltoids using the same amount of memory.

On the other hand, SeqHash has smaller memory and computational costs in the Detection step than reversible sketch, whose complexity is sub-linear of the original key space (with $O(N^{1/\log \log N} \log \log N)$ in memory and $O(N^{3/\log \log N} \log \log N)$ in computation). The main reason is that SeqHash enumerates sub-keys of the heavy keys using multiple intermediate steps, rather than using one step as in reversible sketch. The flip side is that SeqHash requires more memory accesses than reversible sketch. Nevertheless, should the number of memory accesses be a bottleneck, we can reduce this cost by increasing the memory usage (i.e., increasing K), and hence we can reduce the total number of hash tables used by SeqHash for a fixed error rate (see analysis in Section 3).

5.5. Extension to heavy changer detection

In heavy changer detection, we use the notion of legitimate misses to reduce the false negative rate. In this case, we perform more hash operations to determine non-heavy keys as opposed to heavy hitter detection. In this subsection, we provide an upper bound on the computational cost of the Detection step.

Table 2
Complexity results of reversible sketch, deltoids, and SeqHash (the results of reversible sketch and deltoids are obtained by Schweller et al. [17]).

	Update step		
	Memory	Memory accesses	Operations
(a) Update costs			
Reversible sketch	$\Theta\left(\frac{(\log N)^{\Theta(1)}}{\log \log N}\right)$	$\Theta\left(\frac{(\log N)}{\log \log N}\right)$	$\Theta(\log N)$
Deltoids	$\Theta(H \log N)$	$\Theta(\log N)$	$\Theta(\log N)$
SeqHash	$\Theta(H \log N)$	$\Theta(\log N)$	$\Theta(\log N)$
	Memory		Operations
(b) Detection costs			
Reversible sketch	$\Theta\left(N^{\frac{1}{\log \log N}} \cdot \log \log N\right)$		$O\left(HN^{\frac{3}{\log \log N}} \cdot \log \log N\right)$
Deltoids	$\Theta(H \log N)$		$O(H \log N)$
SeqHash	$\Theta(H \log N)$		$\Theta(H \log N)$

Suppose that we fix the design parameters $\alpha = 9$, $b_1 = 16$, and $b_i = 2$ for $i \geqslant 2$ as in our previous complexity analysis. From both Algorithm 2 and Fig. 5, the maximum number of hash operations of the Detection step is

$$\begin{aligned}\text{Computation}_{\max} &= \sum_{i=1}^{D} |\mathcal{C}_{i-1}| \times 2^{b_i} \times M_i \\ &= 2^{b_1} M_1 + (1+\alpha) H \times 2^2 \sum_{i=2}^{D} M_i \\ &\leqslant \max(2^{16}, 40H) \sum_{i=1}^{D} M_i. \end{aligned} \tag{8}$$

If we use a total number of $\sum_{i=1}^{D} M_i = \Theta(\log N)$ hash tables as in heavy hitter detection, then the total number of hash operations is upper bounded by $\Theta(H \log N)$. Although the complexity has a higher multiplicative constant than the case of heavy hitter detection, it remains logarithmic in N and is still less than the Detection step of reversible sketch (see Table 2).

In general, for heavy change detection, we also need more memory to mitigate the false negative rate (see Section 3). In our experiments, we achieve this by increasing K, as well as using estimation that is explained in Section 6.

6. Estimating values of heavy keys using linear regression

In this section, we present a maximum-likelihood-based method that uses a linear regression model to estimate the values of heavy key values that have been recovered from SeqHash. Estimation of values of heavy keys is important for two reasons. First, when the number of heavy keys is large, it is desirable to highlight the most important heavy keys with the highest values. Second, using the estimated values, we can further reduce the false positive rate by eliminating those non-heavy keys included in the candidate set of heavy keys. In the experimental studies in Section 7, we will show that by using estimation, we can reduce the false positive rate significantly at the expense of only a small increase in the false negative rate.

It is important to note that SeqHash presented earlier does not fully utilize the information in the counter values, as all it does is a simple threshold test. Also, it does not take into account the noise values due to non-heavy keys that are determined by the underlying traffic behavior. We now show how we exploit the information of the counter values and the traffic behavior to develop our estimation methods.

Suppose that we are given a candidate set $\mathcal{C}$ of heavy keys, and L heavy buckets, each of which is associated with at least one candidate heavy key in $\mathcal{C}$. Let Y be a vector of length L representing the counter values (or change in counter values for heavy changer detection) for the L heavy buckets. Let A be an $L \times |\mathcal{C}|$ 0–1 matrix in which element A_{ij} equals 1 if the ith heavy bucket is associated with the jth candidate heavy key, or 0 otherwise. Let V be a vector of length $|\mathcal{C}|$ representing the values of heavy keys, and δ be a vector of length L that denote the values due to remaining non-heavy keys to the heavy buckets. Note that both V and δ are the unknown vectors that need to be estimated. Now we can write

$$Y = AV + \delta. \tag{9}$$

In the following, we discuss the model choice for δ in both heavy hitter and heavy changer cases, and present a maximum likelihood estimator for V.

6.1. Heavy hitter estimation

Based on the empirical studies of real traces, for heavy hitter estimation, we find that the distribution of δ is well approximated by a Weibull distribution with mean θ and shape parameter β. Fig. 8 shows Weibull-QQplot of the observed δ distribution for the detection of at most 500 heavy hitters in a real trace studied later in Section 7, using a hash array with $M = 33$ tables and $K = 722$ buckets per table. It is easy to see that the Weibull distribution gives an excellent approximation as a straight line indicates an exact Weibull distribution.

We observe that the shape parameter β is close to 1. When the shape parameter is 1, a Weibull distribution is reduced to an exponential distribution. In this case, the maximum likelihood estimate $\widehat{V}_{\text{MLE}}$ is equivalent to solving the following linear programming problem with respect to V:

$$\text{maximize} \sum_{l=1}^{L} A_l V \text{ subject to } (y_l - A_l V) \ \geqslant \ 0, \tag{10}$$

where y_l is the l-th element of Y and A_l is the l-th row of A.

A computationally cheaper estimator of V, the *countmin* estimator, has been proposed in [4]. The *countmin* estimator for the value of a candidate heavy hitter key is essentially the minimum of all bucket values of y that contain the candidate key. It is straightforward to show that if all the heavy buckets contains exactly one heavy hitter, the maximum likelihood estimator $\widehat{V}_{\text{MLE}}$ reduces to the *countmin* estimator $\widehat{V}_{\min}$. However, from Lemma 1, this is not true in general as only around 70% of the heavy buckets contain exactly one heavy hitter when $H/K = \log 2$. In addition, it can be shown that both $\widehat{V}_{\min}$ and $\widehat{V}_{\text{MLE}}$ have some small positive bias, which is approximately

$$\text{bias} \ \approx \ E\Big[\min_m \widetilde{Y}_m\Big], \tag{11}$$

where $\widetilde{Y}_m$ corresponds to some value of a non-heavy bucket in hash table m in our multi-hash-array structure (note that $1 \leqslant m \leqslant \sum_{i=1}^{D} M_i$). The bias can be approximated accurately using a nonparametric method as follows. First, we randomly select a non-heavy bucket from each hash table and compute the minimum value among the selected non-heavy buckets. Then we repeat this process many times and compute the empirical mean of the minimum values.

In summary, our estimation method is based on linear regression, by solving the linear programming problem in (10). The result is then corrected with the bias in (11).

In addition to the *countmin* estimator, a least-square estimator of the heavy hitters has been proposed in [11], whose approach can be viewed as the maximum likelihood

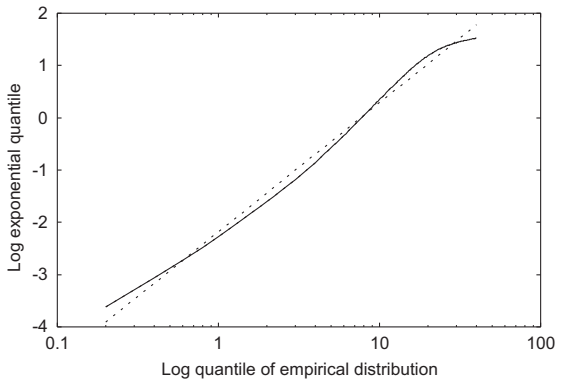


Fig. 8. The error distribution of bucket values in a hash array for heavy hitter detection with $K = 722$, $M = 33$, and 500 heavy hitters (see Experiment 1 in Section 7). The dotted line indicates the true Weibull distribution.

estimate when the error distribution follows a normal distribution. In Section 7, we compare our approach with the least-square method.

6.2. Heavy changer estimation

Based on the empirical studies of real traces, we find the distribution of δ in the heavy changer case is well approximated by a double exponential distribution. In such case, the maximum likelihood estimate $\widehat{V}_{\text{MLE}}$ for the linear regression problem in (9) can be obtained from solving a L_1-regression problem. In particular, when all the heavy buckets contain exactly one heavy hitter, then $\widehat{V}_{\text{MLE}}$ corresponds to the median estimator, similar to that proposed in [9]. The median estimator for the value of a candidate key is the median of all bucket values that contain the candidate key.

To illustrate the benefits of using estimation, we implement heavy changer estimation by directly applying the heavy hitter estimation approach. We find that this approximation can provide substantial improvement over the heavy changer detection without estimation.

7. Experimental studies

When non-heavy key values are no longer negligible as is often the case in real traces, some of the non-heavy buckets will be considered to be heavy, leading to more false positives. Therefore, we need additional memory to counter this noise effect. In this section, we use trace-driven simulation to study how various choices of parameters tolerate the presence of noise.

Using Internet traces captured from various sources, we evaluate SeqHash in identifying heavy keys (i.e., heavy hitters and heavy changers) when non-heavy keys have non-negligible values. We mainly our scheme with Deltoids [7] using its publicized software. In addition, we analyze the improvement of SeqHash when it is coupled with linear regression presented in Section 6. In addition, we compare SeqHash with the modular hashing scheme that is used in reversible sketch [17] (see Experiment 7).

7.1. Traces

The results presented here are based on a 1-h uni-directional trace from NLANR [14]. The trace contains about 50 GB of Internet traffic collected from 10:00 pm to 11:00 pm on June 1, 2004 at an OC-192 link connecting between Indianapolis and Kansas city in the United States. The huge volume of collected traffic allows us to demonstrate the effectiveness of SeqHash in a high speed network. We repeat our evaluation using the NLANR traces collected from the same source but at other times as well as using private traces collected at an OC-48 link of an ISP, and similar results are observed.

We divide the 1-h NLANR trace into six 10-min monitoring intervals. For heavy hitter detection, we identify the source IPs whose data volume exceeds a threshold in each interval. We then average the results across all intervals. On the other hand, for heavy changer detection, we identify the source IPs whose absolute change of data volume is above a threshold in each pair of adjacent intervals. We then average the results across all pairs of adjacent intervals. It should be noted that the length of a monitoring interval varies across applications, and our goal here is to evaluate the effectiveness of the heavy key detection approaches. Evaluation of different lengths of monitoring intervals can be found in [15].

7.2. Experiment setup

Unless otherwise stated, our discussion focuses on the 32-bit key space based on source IP addresses. However, we also experiment the 64-bit key space defined by source–destination IP addresses.

In our experiments, we assume the maximum number of heavy keys (i.e., H) is 500. Also, we vary K (i.e., the number of buckets in each hash table) and M (i.e., $\sum_{i=1}^{D} M_i$, the total number of hash tables across all hash arrays) for identifying at most H heavy keys. For evaluation purpose, we select different thresholds, each of which corresponds to a true number of heavy keys. We therefore maintain a baseline structure that keeps track of the per-key data volume for such threshold selection. The baseline structure is also used for assessing the accuracies of recovering heavy keys of SeqHash and Deltoids.

Our implementation is written in C. In our prior conference version [3], we use MD5 for our hash functions. In this journal version, we revise our hash function implementation using the one in Snort [18]. While both Snort-based and MD5-based hash functions lead to very similar accuracy results in our experiments, the former requires significantly less time in update steps (see Experiment 6). In addition, we use the freely distributed package *lp_solve* [12] to implement linear programming (LP) for linear regression. Our experiments are conducted on a machine with CPU speed 2.8 GHz and DRAM memory.

7.3. Metrics

We are mainly interested in two accuracy metrics: (1) *false positive rate*, defined as the ratio of the number of non-heavy keys to the number of keys returned, and (2)

false negative rate, defined as the ratio of the number of true heavy keys that are not returned to the number of true heavy keys. In addition, we also consider the *update time* and *detection time* of executing SeqHash as well as the *estimation errors* in estimating the values of the recovered heavy keys.

Experiment 1 (*Analysis of finding heavy hitters (without linear regression)*). To counter the noise effect, we need additional memory by increasing K and/or M for successful heavy key detection. Thus, we study the impact with different choices of K and M. We begin our analysis by first excluding linear regression described in Section 6.

As shown in Section 5, if $H = 500$ and non-heavy keys have negligible values, then we can choose $K = H/\log 2 \approx 722$, and $M = 33$ where $M_1 = 4$, $M_2 = \cdots = M_8 = 2$, and $M_9 = 15$ (i.e., 9 hash arrays) for SeqHash. This is the "noise-free" configuration. To counter the effects of noise due to non-heavy keys, we increase the memory by 50% and 100% by using different values of K and M shown in Table 3 (where each counter is assumed to be of size 4 bytes).

Fig. 9 shows the false positive rate of finding heavy hitters using SeqHash (note that since every bucket that contains heavy hitters must be a heavy bucket, there is no false negative). With the original noise-free configuration $K = 722$ and $M = 33$, the false positive rate can be as high as 32%. However, by increasing the number of counters, we can reduce the false positive rate significantly to less than 6% by increasing K by 50% (for $M = 33$ and $K = 1083$) and further to less than 3% by doubling K (for $M = 33$ and $K = 1444$).

Table 3
Configurations of K and M.

K	M	$(M_1, M_{2\leqslant i\leqslant 8}, M_9)$	Number of counters (memory size)
722	33	(4, 2, 15)	23826 (93 KB)
722	50	(6, 3, 23)	36100 (141 KB)
722	66	(8, 4, 30)	47652 (186 KB)
1083	33	(4, 2, 15)	35739 (140 KB)
1444	33	(4, 2, 15)	47652 (186 KB)

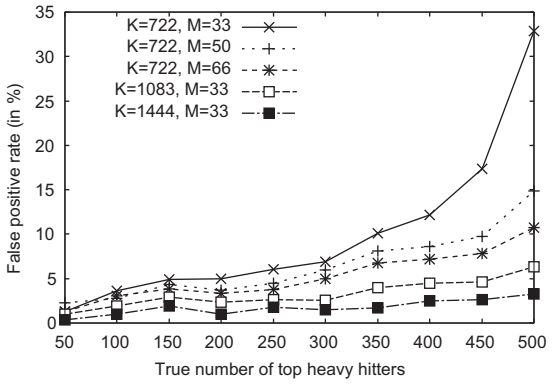


Fig. 9. Experiment 1: false positive rate of finding heavy hitters using SeqHash without linear regression.

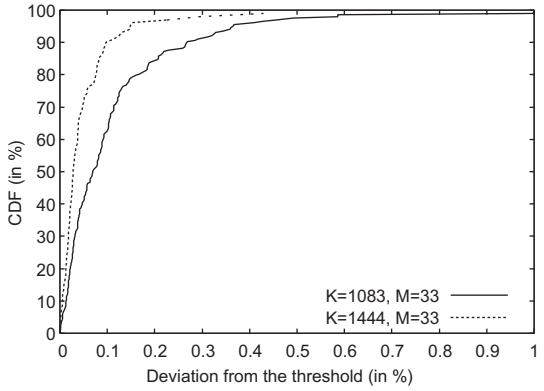


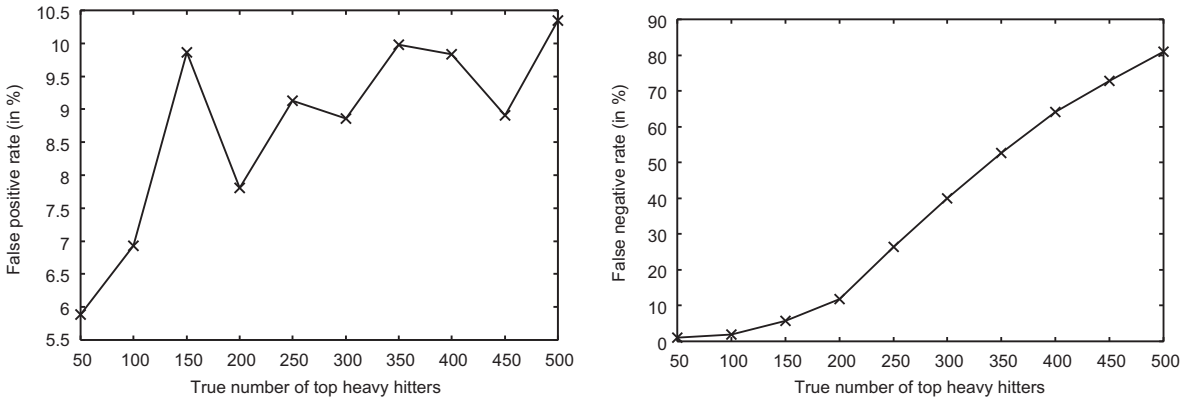
Fig. 10. Experiment 1: deviations of false positives from the threshold for finding the top 500 heavy hitters.

In the presence of noise, we note that increasing K is more advantageous than increasing M. Intuitively, as K increases, the values of non-heavy keys are distributed across more buckets. Thus, a bucket that contains only non-heavy keys is less likely to become a heavy bucket, leading to a reduced false positive rate. Also, increasing M is less desirable in practice because it increases the number of hash operations needed to record keys into the hash arrays.

To further examine the values of the false positives, Fig. 10 depicts the percentage of deviations of these values with respect to the threshold for the case of finding the top heavy hitters using the configurations with $M = 33$ and $K = 1083$ and 1444. In fact, most of the false positives do not actually deviate much from the threshold. For instance, the proportion of false positives that have values within 15% of the threshold is more than 80% when $M = 33$ and $K = 1083$, and achieves 100% when $M = 33$ and $K = 1444$. It shows that the heavy hitter candidates returned from SeqHash can effectively approximate the set of true heavy hitters.

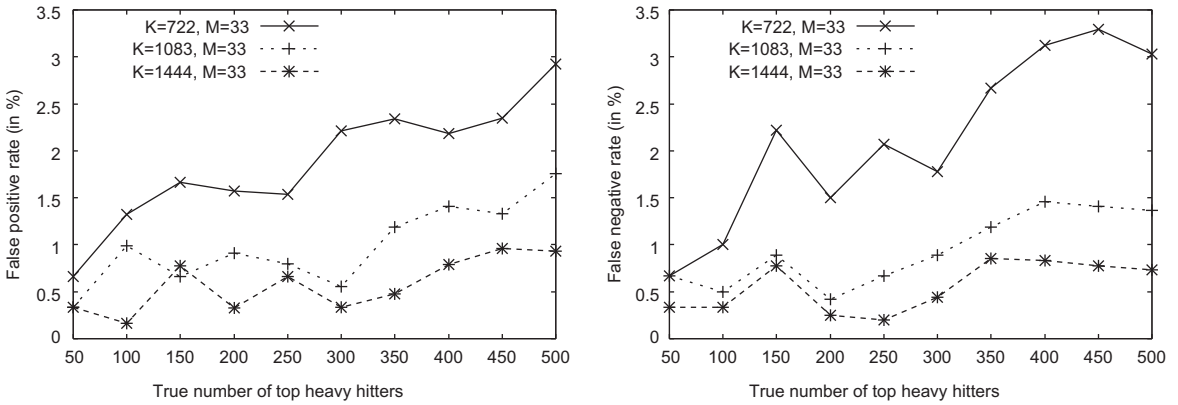
We now compare SeqHash with Deltoids using its publicized software. Fig. 11 shows the accuracy of using Deltoids to identify heavy hitters. Here, we set the number of hash tables and the number of buckets in each hash table to be 4 and 361, respectively. Since each of its buckets is associated with $\log_2 N + 1 = 33$ counters, its total number of counters is no less than all of our configurations. While Deltoids has less than 10% of false positive rate, its false negative rate is significantly high (up to 80%) as more heavy hitters need to be identified, meaning that many true heavy hitters evade detection. We have also tried other combinations of the numbers of hash tables and buckets in each hash table with the same total number of counters, but the false negative rate remains significantly high. In short, with the same or even less amount of memory, SeqHash provides a much more accurate heavy hitter detection than does Deltoids.

Experiment 2 (*Analysis of finding heavy hitters (with linear regression)*). Here, we analyze how the heavy hitter detection benefits from linear regression presented in Section 6.



(a) False positive rate (b) False negative rate

Fig. 11. Experiment 1: accuracy of deltoids in finding heavy hitters.



(a) False positive rate (b) False negative rate

Fig. 12. Experiment 2: accuracy of finding heavy hitters with linear regression.

As Experiment 1 shows that increasing K outperforms increasing M, we focus on the configurations with $M = 33$, and $K = 722$, 1083, and 1444.

Fig. 12 shows the accuracy of finding heavy hitters when we couple SeqHash with linear regression. Referring to Fig. 9 in Experiment 1, for $K = 722$ and $M = 33$, the false positive rate for identifying 500 heavy hitters is almost 32% without linear regression. However, linear regression reduces this false positive rate to less than 3%, while introducing a false negative rate 3.3% (i.e., the total error rate is about 6%). Also, for $K = 1083$ and $M = 33$, the false positive rate for identifying 500 heavy hitters is also about 6% when no linear regression is used. This shows that linear regression can reduce the amount of memory required to achieve the same total error rate.

In terms of the accuracy of estimation, we show that linear regression provides a better estimate of values of heavy hitters than does the least-square method proposed in [11]. Here, we consider the following two error measures:

$$\mathrm{Err}_1 = \frac{1}{|\mathcal{C}|}\sum_{x\in\mathcal{C}} |v_x^{\mathrm{est}} - v_x|,$$

$$\mathrm{Err}_2 = \sqrt{\frac{1}{|\mathcal{C}|}\sum_{x\in\mathcal{C}} \left(v_x^{\mathrm{est}} - v_x\right)^2},$$

where v_x and v_x^{est} are respectively the true value and the corresponding estimate of key x, and $\mathcal{C}$ is the final candidate set returned from SeqHash.

Table 4 shows the error measures (in unit MB) of both linear and least-square regressions in estimating the data volumes of the top 500 heavy hitters. It shows that linear regression always outperforms least-square regression in both error measures. We have also tried other types of error measures and linear regression still provides better results.

The improvement of our linear regression method is a result of modeling the characteristics of non-heavy keys in data traffic. While by no means do we claim our linear regression approach is the best choice for all instances of

Table 4
Experiment 2: accuracies of linear and least-square regressions.

Configuration	Linear		Least-square	
	Err_1	Err_2	Err_1	Err_2
$K = 722$, $M = 33$	0.7381	1.0233	2.1737	2.8462
$K = 1083$, $M = 33$	0.3950	0.5469	0.7658	0.9897
$K = 1444$, $M = 33$	0.2146	0.3035	0.3992	0.5316

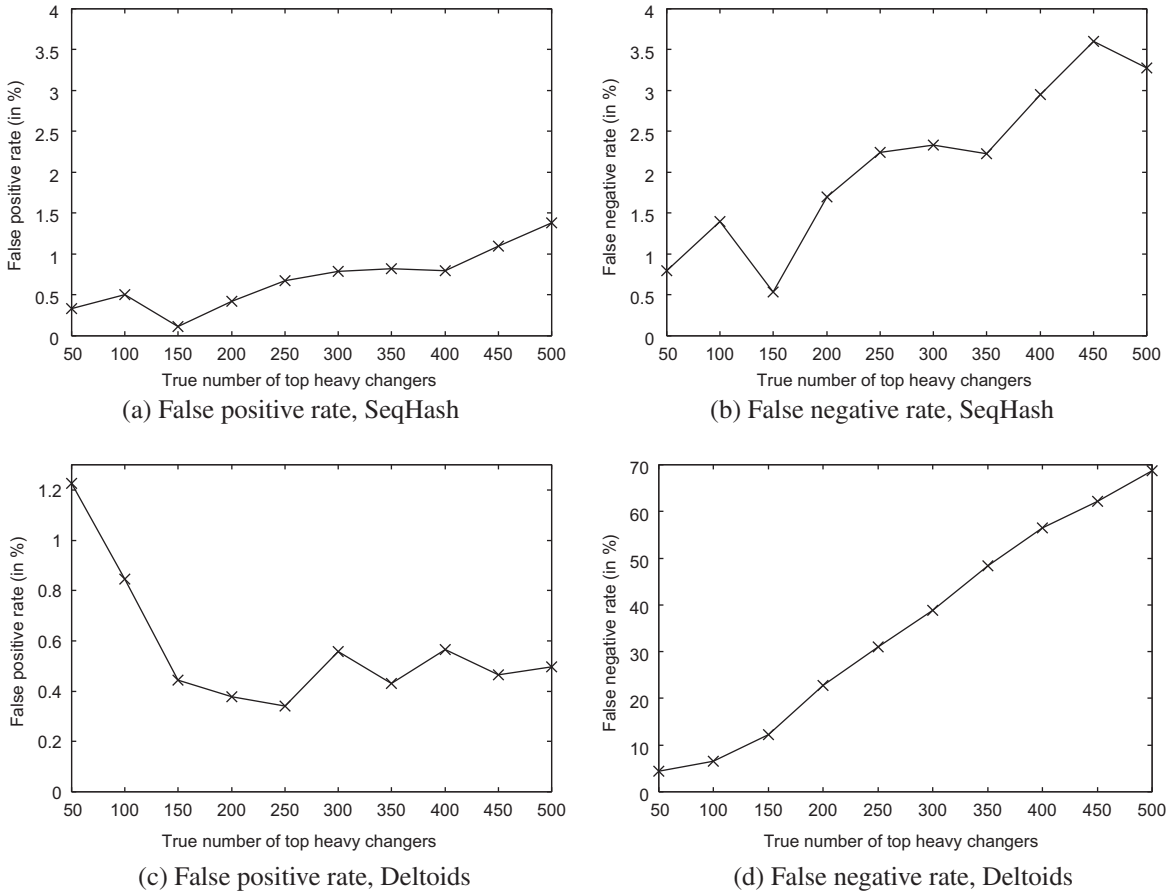
traffic traces, our analysis provides insights on how heavy key detection techniques can benefit from understanding the behavior of data traffic.

Experiment 3 (*Accuracy of finding heavy changers*). We now compare both SeqHash (with linear regression) and Deltoids in heavy changer detection. Since the positive and negative changes can cancel each other, some of the buckets that contain heavy changers will not be identified as heavy buckets. Therefore, we need even more memory to mitigate this impact. Here, for SeqHash, we consider the configuration with $K = 1444$ and $M = 33$, while for Deltoids, we use the same configuration as in Experiment 1. Thus, both approaches are allocated with the same number of counters.

Fig. 13 depicts the accuracy of finding heavy changers both schemes. While Deltoids only has at most 1.2% false positive rate, its false negative rate can be as high as 70%. On the other hand, with the same number of counters, SeqHash bounds the false positive and negative rates within 1.4% and 3.6%, respectively.

Experiment 4 (*Analysis of 64-bit key space*). We further evaluate SeqHash (with linear regression) using the 64-bit source–destination IP pairs as the key space. We consider a special case with $M = 66$ and 25 hash arrays, such that $M_1 = 4$, $M_2 = \cdots = M_{24} = 2$, $M_{25} = 16$. For the case of finding heavy hitters, we start with $M = 66$ and $K = 1444$. Fig. 14(a) and (b) show the accuracy of finding heavy hitters. The average false positive and negative rates are no more than 1.6% and 1%, respectively. On the other hand, for the case of finding heavy changers, we set $K = 1800$ to further counter the cancellation of positive and negative changes. Fig. 14(c) and (d) show the accuracy of finding heavy changers. The average false positive and negative ratios are no more than 0.9% and 2.4%, respectively.

Note that we observe variations of the false positive/negative rates in the figures. For example, Fig. 14(d) shows "dips" when the true number of heavy changers changes from 50 to 100 and from 400 to 450. This is mainly due to



(a) False positive rate, SeqHash
(b) False negative rate, SeqHash
(c) False positive rate, Deltoids
(d) False negative rate, Deltoids

Fig. 13. Experiment 3: accuracy of finding heavy changers.

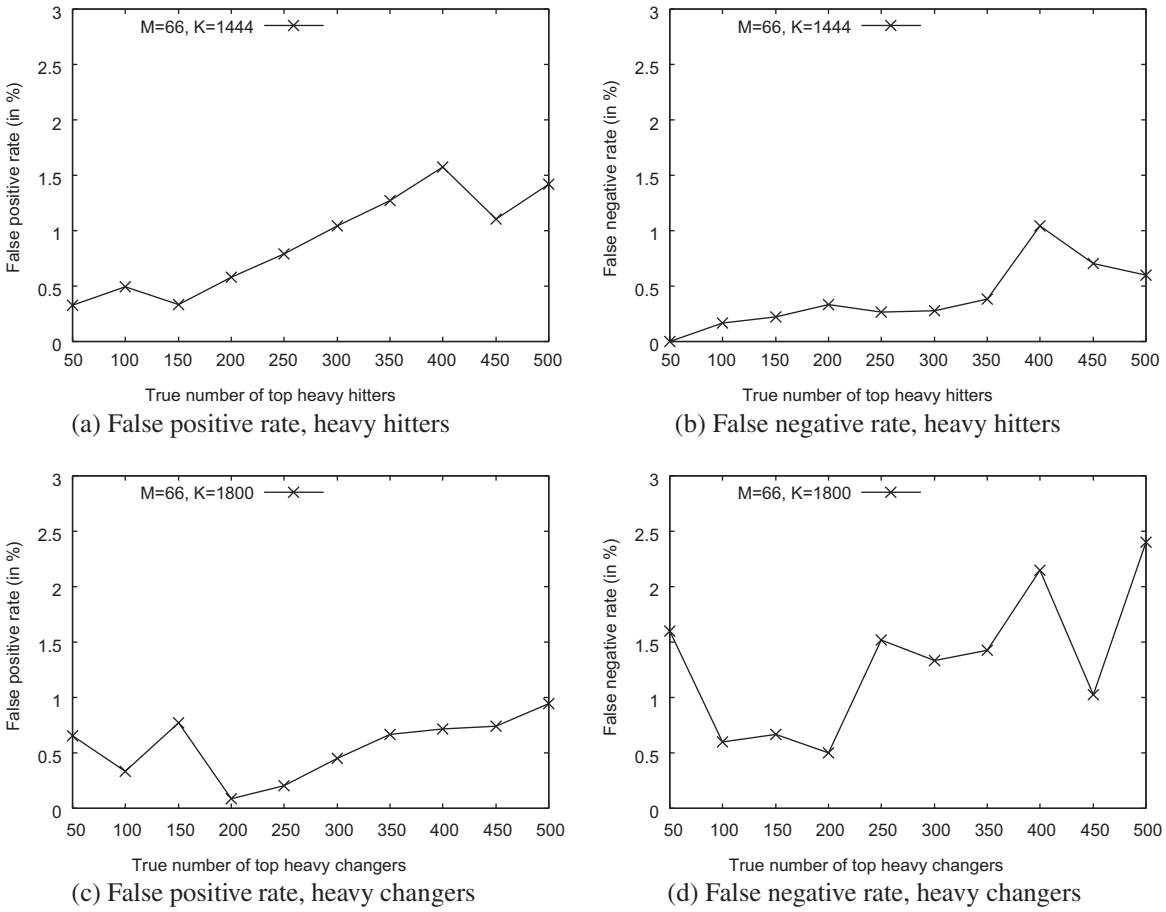


Fig. 14. Experiment 4: accuracy of SeqHash in finding the top 500 heavy hitters/changers of 64-bit source–destination IP pairs.

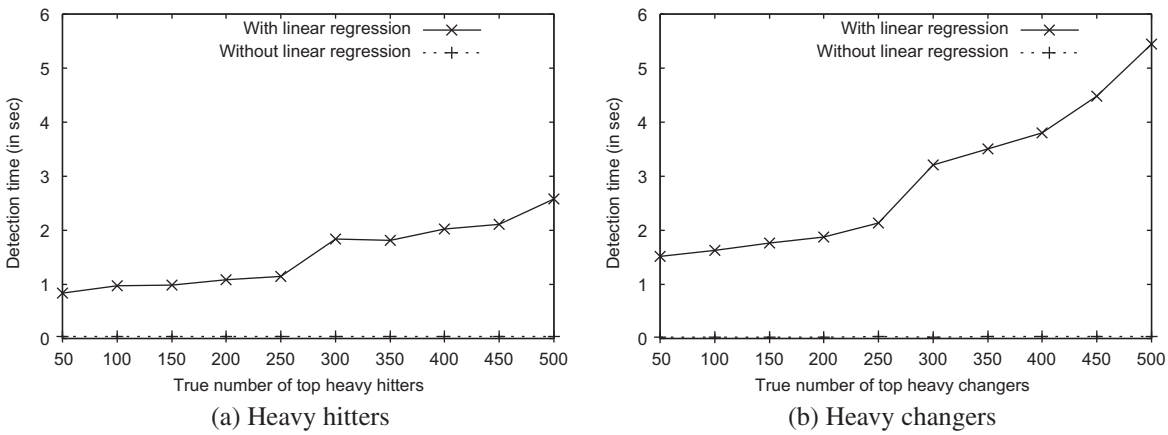


Fig. 15. Experiment 5: detection time of SeqHash of recovering the top 500 heavy hitters/changers (with linear regression).

the statistical variations of our real traffic trace data, such that a slight change of the number of false positive/negative keys can affect the false positive/negative rates. Nevertheless, the number of false positive/negative keys remains small in all cases. For instance, when the true number of heavy changers is between 50 and 200, the expected number of false negative keys is less than one (Fig. 14(d)).

Experiment 5 (*Detection time*). This experiment evaluates the execution time of recovering heavy keys. We consider the case where $M = 33$ and $K = 1444$, and enable linear regression. Fig. 15(a) and (b) show the detection times of recovering heavy hitters and heavy changers, respectively. The detection times for finding heavy hitters and heavy changers, when linear regression is used, are within 2.6 and 5.5 s, respectively. We point out that the execution time of SeqHash is less than 0.05 s if linear regression is not used. Therefore, the major overhead is on solving the LP problem for linear regression. We expect that the detection time can be further improved with more computationally efficient hash functions and LP packages.

Experiment 6 (*Update time*). We now evaluate the speed of the update step of SeqHash using the Snort-based hash function implementation [18]. We have SeqHash process the NLANR traces as far as possible on a machine with CPU speed 2.8 GHz and DRAM memory. Fig. 16(a) and (b) show the update times for the 32-bit key space (where we set $M = 33$) and the 64-bit key space (where we set $M = 66$) in each of the six 10-min monitoring intervals, respectively. In each of the monitoring intervals, we observe that SeqHash can update all packets within 110 and 300 s for the 32-bit key space and 64-bit key space, respectively, implying that SeqHash can catch up with the packet rate of the traces that we use in our experiments.

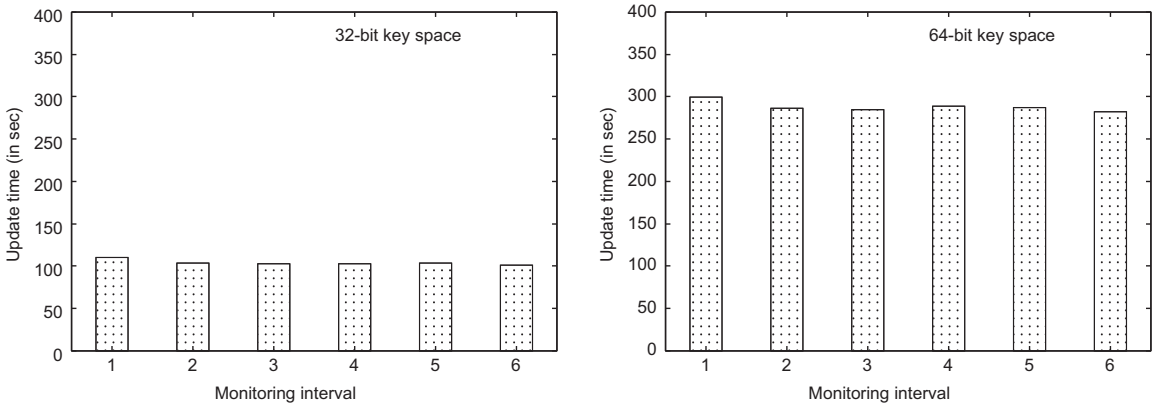
Note that our current implementation runs on a single process. The update speed can be further improved if we can leverage parallel processing on multi-core architectures. In a high level, we can dispatch packets to different cores in a round-robin manner, and each core has a thread that updates packets to its own sketch. We can later combine the sketches from multiple threads into an aggregate sketch by summing the bucket values (note that our data structure fulfills the linearity property [9]), and solve the heave-key detection problem on the aggregate sketch.

Experiment 7 (*Analysis of modular hashing*). We now compare SeqHash with modular hashing [17] (see our discussion in Section 4.4), both of which are possible implementations of multi-level hashing. Here, we focus on identifying the heavy hitters among the 32-bit source IP addresses. Our setting is based on Experiment 1, such that we disable linear regression for sequential hashing. This enables us to analyze the baseline implementations of multi-level hashing.

For modular hashing, we first apply managling to each source IP address to destroy correlations among the keys. Based on [17], we consider a simple mangling function $f(x) = 101^{-1}x(\mathrm{mod}\ 2^{32})$ for input key x. We then divide each 32-bit mangled key into four octets, and apply modular hashing. After the detection step, the returned heavy keys will be unmangled using the function $f^{-1}(x) = 101x(\mathrm{mod}2^{32})$, and we evaluate the accuracy. The configurations for modular hashing are $M = 6$, 9, and 12, and $K = 2^{12} = 4096$. On the other hand, for sequential hashing, we use $M = 33$ and $K = 722$, 1083, and 1444, as in Experiment 1. Our goal is to compare both modular hashing and sequential hashing using the similar amount of memory.

Fig. 17 shows the false positive rates of finding heavy hitters in both sequential hasing and modular hashing under different configurations (note that we do not have false negatives as linear regression is disabled). Note that modular hashing has a significantly high false positive rate when $M = 6$. When we increase the memory size, the false positive rate of modular hashing decreases, but remains larger than that of SeqHash when the true number of top heavy hitters is high.

Here, we do not claim modular hashing is less accurate, as the choice of the mangling function could affect the accuracy. In [17], a more sophisticated mangling function based on Galois Field operations is proposed, and it is shown to be very effective in destroying correlations of keys. The study of the impact of mangling functions on multi-level hashing is beyond the scope of our work.



(a) 32-bit keys, $M = 33$ (b) 64-bit keys, $M = 66$

Fig. 16. Experiment 6: update time of SeqHash.

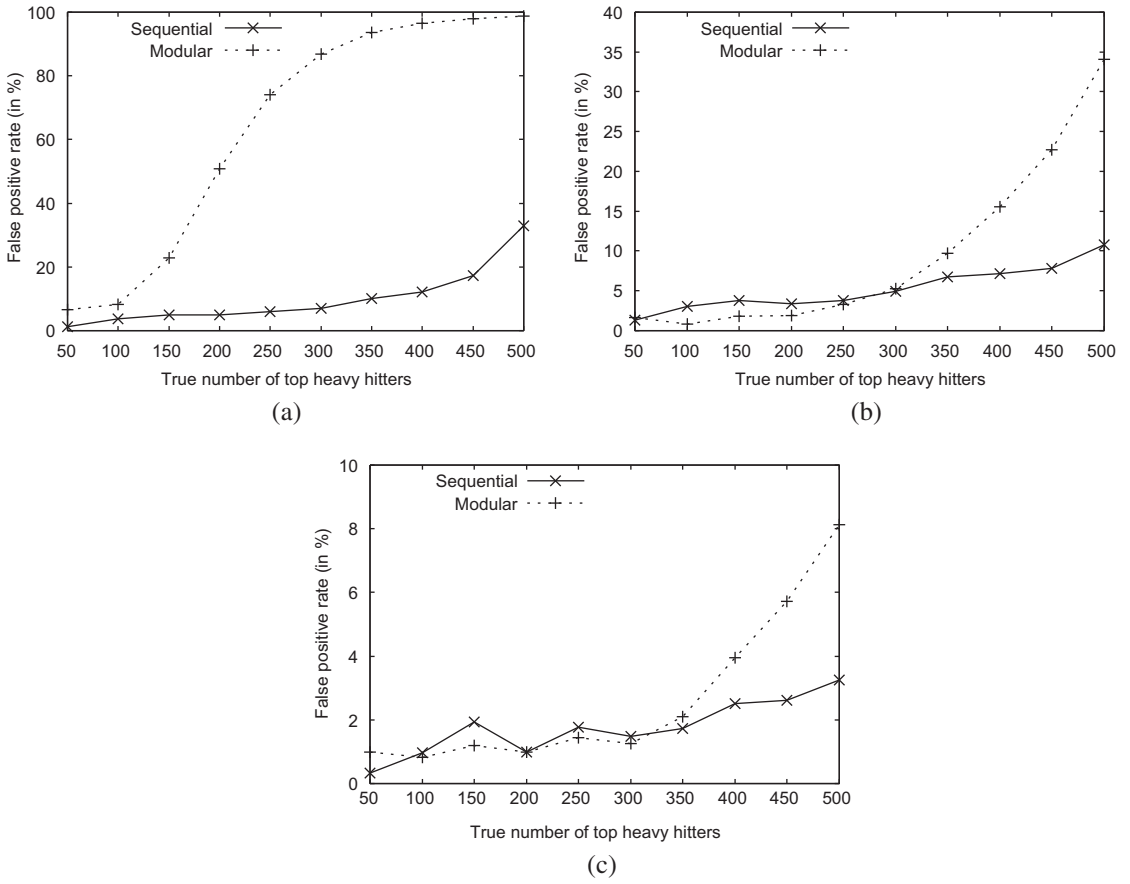


Fig. 17. Experiment 7: comparison of modular hashing and sequential hashing: (a) modular hashing, $M = 6$, $K = 4096$, sequential hashing, $M = 33$, $K = 722$; (b) modular hashing, $M = 9$, $K = 4096$, sequential hashing, $M = 33$, $K = 1083$; (c) Modular hashing, $M = 12$, $K = 4096$, sequential hashing, $M = 33$, $K = 1444$. The results for sequential hashing are identical to those in Experiment 1, Fig. 9.

7.4. Summary

Using a memory-efficient data structure, we show that SeqHash provides more accurate heavy hitter and heavy changer detection than does Deltoids in the presence of noise. With linear regression, the accuracy of SeqHash is further improved. Moreover, we show that SeqHash allows fast detection and supports large key space.

8. Conclusion

In this paper, we consider how to identify the keys (e.g., IPs or flows) that have large data volume or large volume change in a high speed network. Given the infeasibility of tracking all keys, we first derive the lower-bound memory requirement for recovering heavy keys with respect to a fixed false positive rate. We then propose SeqHash, which uses a sketch data structure to achieve accurate and fast identification of heavy keys. We show that with different choices of design parameters, we can readily achieve a trade-off between memory usage and computational overhead. In addition, we propose a linear-regression-based method to accurately estimate the values of heavy keys and to further improve the accuracy of heavy key detection. Finally, we show via extensive trace-driven simulation that SeqHash is more robust in identifying heavy keys as compared to the Deltoids approach.

Note: An earlier and shorter conference version of this paper appeared in IEEE INFOCOM '07 [3]. We make several extensions in this journal version. First, we formally derive the costs of memory usage and computational cost of our proposed scheme (see Section 5). We revise our evaluation using a fast hash function to minimize the monitoring and detection times (see Section 7). We also present more rigorous arguments when comparing our proposed scheme with previous work.

Acknowledgment

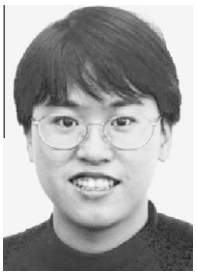
The work of Patrick P.C. Lee was supported in part by the CUHK faculty direct grant (project number: 2050447).

References

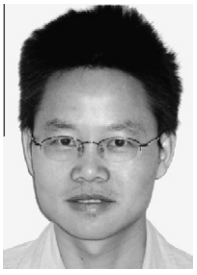
[1] B. Bloom, Space/time trade-offs in hashing coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.
[2] A. Broder, M. Mitzenmacher, Network applications of bloom filters: a survey, Internet Mathematics 1 (4) (2003) 485–509.

[3] T. Bu, J. Cao, A. Chen, P.P.C. Lee, A fast and compact method for unveiling significant patterns in high speed networks, in: Proceedings of IEEE INFOCOM, 2007.
[4] G. Cormode, F. Korn, S. Muthukrishnan, D. Srivastava, Finding hierarchical heavy hitters in data streams, in: VLDB, 2003.
[5] X. Dimitropoulos, P. Hurley, A. Kind, Probabilistic lossy counting: an efficient algorithm for finding heavy hitters, ACM SIGCOMM Computer Communication Review 38 (1) (2008).
[6] C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the Elephants, ignoring the Mice, ACM Transactions on Computer Systems 21 (3) (2003) 270–313.
[7] G. Cormode, S. Muthukrishnan, What's new: finding significant differences in network data streams, in: Proceedings of IEEE INFOCOM, 2004.
[8] M. Kodialam, T. Lakshman, S. Mohanty, Runs based traffic estimator (RATE): a simple, memory efficient scheme for per-flow rate estimation, in: Proceedings of IEEE INFOCOM, 2004.
[9] B. Krishnamurthy, S. Sen, Y. Zhang, Y. Chen, Sketch-based change detection: methods, evaluation, and applications, in: Internet Measurement Conference, 2003.
[10] A. Kumar, J. Xu, J. Wang, O. Spatschek, L. Li, Space-code bloom filter for efficient per-flow traffic measurement, in: Proceedings of IEEE INFOCOM, 2004.
[11] G.M. Lee, H. Liu, Y. Yoon, Y. Zhang, Improving sketch reconstruction accuracy using linear least squares method, in: Internet Measurement Conference, 2005.
[12] lp_solve. <http://groups.yahoo.com/group/lp_solve/>.
[13] G. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of the VLDB, 2002.
[14] NLANR. Abilene-III Trace Data. <http://pma.nlanr.net/Special/ipls3.html>.
[15] K. Papagiannaki, R. Cruz, C. Diot, Network performance monitoring at small time scales, in: IMC, 2003.
[16] V. Paxson, R. Sommer, N. Weaver, An architecture for exploiting multi-core processors to parallelize network intrusion prevention, in: IEEE Sarnoff Symposium, 2007.
[17] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, E. Parsons, Y. Zhang, P. Dinda, M. Kao, G. Memik, Reversible sketches: enabling monitoring and analysis over high-speed data streams, IEEE/ACM Transactions on Networking 15 (5) (2007).
[18] Snort. <http://www.snort.org/>.

Tian Bu received his Ph.D. in Computer Science from University of Massachusetts, Amherst in 2002. He has been at Bell Labs, Alcatel-Lucent since 2002. He is now the CTO of a wireless data network security and management project at Alcatel-Lucent. His current research includes network security and network modeling and performance evaluation.



Jin Cao has been a member of technical staff at Bell Laboratories since 1997. She got her Ph.D. from the Department of Mathematics and Statistics, McGill University, Canada in 1997 and joined Bell Laboratories later in the year. Her Ph.D. thesis was on the statistical analysis of brain images. Her current research focuses on statistical problems arising from data networks, for example, network tomography, traffic modeling and simulation, performance analysis, and data streaming algorithms.



Aiyou Chen received the B.S. degree in mathematics from Wuhan University, Wuhan, China, in 1997, the M.S. degree in probability and mathematical statistics from Peking University, Beijing, China, in 2000, and the Ph.D. degree in statistics from University of California, Berkeley, in 2004. He is currently a Member of Technical Staff with Bell Laboratories, Alcatel Lucent, Murray Hill, NJ. His current research interests include statistical learning, social network models, streaming data, and statistical inference in network applications.

Patrick P.C. Lee received the B.E. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an assistant professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in network robustness and security.