Reverse Hashing for High-speed Network Monitoring: Algorithms, Evaluation, and Applications

Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta,
Yin Zhang†, Peter Dinda, Ming-Yang Kao, Gokhan Memik
Department of Electrical Engineering and Computer Science,
Northwestern University, Evanston, IL 60208
†Department of Computer Science, University of Texas at Austin, Austin, TX 78712
{schwellerr, lizc, ychen, ygao, ashish, pdinda, kao}@cs.northwestern.edu,
yzhang@cs.utexas.edu, memik@ece.northwestern.edu

***Abstract*—**

A key function for network traffic monitoring and analysis is the ability to perform aggregate queries over multiple data streams. Change detection is an important primitive which can be extended to construct many aggregate queries. The recently proposed sketches [1] are among the very few that can detect heavy changes online for high speed links, and thus support various aggregate queries in both temporal and spatial domains. However, it does not preserve the keys (*e.g.*, source IP address) of flows, making it difficult to reconstruct the desired set of anomalous keys. In an earlier abstract we proposed a framework for a *reversible* sketch data structure that offers hope for efficient extraction of keys [2]. However, this scheme is only able to detect a single heavy change key and places restrictions on the statistical properties of the key space.

To address these challenges, we propose an efficient *reverse hashing* scheme to infer the keys of culprit flows from reversible sketches. There are two phases. The first operates online, recording the packet stream in a compact representation with negligible extra memory and few extra memory accesses. Our prototype single FPGA board implementation can achieve a throughput of over 16 Gbps for 40-byte-packet streams (the worst case). The second phase identifies heavy changes and their keys from the representation in nearly real time. We evaluate our scheme using traces from large edge routers with OC-12 or higher links. Both the analytical and experimental results show that we are able to achieve online traffic monitoring and accurate change/intrusion detection over massive data streams on high speed links, all in a manner that scales to large key space size. To the best of our knowledge, our system is the *first* to achieve these properties simultaneously.

I. Introduction

The ever-increasing link speeds and traffic volumes of the Internet make monitoring and analyzing network traffic a challenging but essential service for managing large ISPs. A key function for network traffic analysis is the ability to perform aggregate queries over multiple data streams. This aggregation can be either temporal or spatial. For example, consider applying a time series forecast model to a sequence of time intervals over a given data stream for the purpose of determining which flows are exhibiting anomalous behavior for a given time interval. Alternately, consider a distributed detection system where multiple data streams in different locations must be aggregated to detect distributed attacks, such as an access network where the data streams from its multiple edge routers need to be aggregated to get a complete view of the traffic, especially when there are asymmetric routings.

Meanwhile, the trend of ever-increasing link speed motivates three highly desirable performance features for high-speed network monitoring: 1) a small amount of memory usage (to be implemented in SRAM); 2) a small number of memory accesses per packet [3], [4]; and 3) scalabilty to a large key space size. A network flow can be characterized by 5 tuples: source and destination IP addresses, source and destination ports, and protocol. These add up to 104 bits. Thus, the system should at least scale to a key space of size 2^{104}.

In response to these trends, a special primitive called *heavy hitter detection (HHD)* over massive data streams has received a lot of recent attention [5], [6], [4], [7]. The goal of HHD is to detect keys whose traffic exceeds a given threshold percentage of the total traffic. However, these solutions do not provide the much more general, powerful ability to perform aggregate queries. To perform aggregate queries, the traffic recording data structures must have *linearity*, *i.e.*, two traffic records can be linearly combined into a single record structure as if it were constructed with two data streams directly.

The general aggregate queries can be of various forms. In this paper, we show how to efficiently perform *change detection*, an important primitive which can be extended to construct many aggregate queries. The change detection problem is to determine the set of flows whose size changes significantly from one period to another. That is, given some time series forecast model (ARIMA, Holt-Winters, etc.) [1], [8], we want to detect the set of flows whose size for a given time interval differs significantly from what is predicted by the model with respect to previous time intervals. This is thus a case of performing aggregative queries over temporally distinct streams. In this paper, we focus on a simple form of change detection in which we look at exactly two temporally adjacent time intervals and detect which flows exhibit a large change in traffic between the two intervals. Although simple, the ability to perform this type of change detection easily permits an extension to more sophisticated types of aggregation. Our goal is to *design efficient data structures and algorithms that achieve near real-time monitoring and* $flow - level$ *heavy change detection on massive, high bandwidth data streams,*

and then push them to real-time operation through affordable hardware assistance.

The sketch, a recently proposed data structure, has proven to be useful in many data stream computation applications [6], [9], [10], [11]. Recent work on a variant of the sketch, namely the k-ary sketch, showed how to detect heavy changes in massive data streams with small memory consumption, constant update/query complexity, and provably accurate estimation guarantees [1]. In contrast to the heavy hitter detection schemes, sketch has the linearity properties to support aggregate queries as discussed before.

Sketch methods model the data as a series of (*key, value*) pairs where the key can be a source IP address, or a source/destination pair of IP addresses, and the value can be the number of bytes or packets, *etc.*. A sketch can indicate if any given key exhibits large changes, and, if so, give an accurate estimate of the change.

However, sketch data structures have a major drawback: they are not *reversible*. That is, a sketch cannot efficiently report the set of all keys that have large change estimates in the sketch. A sketch, being a summary data structure based on hash tables, does not store any information about the keys. Thus, to determine which keys exhibit a large change in traffic requires either exhaustively testing all possible keys, or recording and testing all data stream keys and corresponding sketches [3], [1]. Unfortunately, neither option is scalable.

To address these problems, in an earlier extended abstract, we proposed a novel framework for efficiently *reversing* sketches, focusing primarily on the k-ary sketch [2]. The basic idea is to hash intelligently by modifying the input keys and/or hashing functions so that we can recover the keys with certain properties like big changes without sacrificing the detection accuracy. We note that streaming data recording needs to done continuously in real-time, while change/anomaly detection can be run in the background executing only once every few seconds with more memory (DRAM).

The challenge is this: how can we make data recording extremely fast while still being able to support, with reasonable speed and high accuracy, queries that look for heavy change keys? In our prior abstract [2], we only developed the general framework, and focused on the detection of a *single* heavy change which is not very useful in practice. However, *multiple* heavy change detection is *significantly* harder as shown in this paper. Moreover, we address the reversible sketch framework in detail, discussing both the theoretical and implementation aspects. We answer the following questions.

- How fast can we record the streaming traffic, with and without certain hardware support?
- How can we simultaneously detect *multiple* heavy changes from the reversible sketch?
- How can we obtain high *accuracy* and efficiency for detecting *a large number of* heavy changes?
- How can we protect the heavy change detection system from being subverted by attackers (*e.g.*, injecting false positives into the system by creating spoofed traffic of certain properties)?
- How does the system perform (accuracy, speed, *etc.*) with various key space sizes under real router traffic?

In addressing these questions, we make the following contributions.

- For data stream recording, we design improved *IP mangling* and *modular hashing* operations which only require negligible extra memory consumption (4KB - 8KB) and few (4 to 8) additional memory accesses per packet, as compared to the basic sketch scheme. When implemented on a single FPGA board, we can sustain more than 16Gbps even for a stream of 40-byte-packets (the worst case traffic).
- We introduce the *bucket index matrix algorithm* to simultaneously detect multiple heavy changes efficiently. We further propose the *iterative approach* to improve the scalability of detecting a large number of changes. Both space and time complexity are sub-linear in the key space size.
- To improve the accuracy of our algorithms for detecting heavy change keys we apply the following two approaches: 1) To reduce false negatives we additionally detect keys that are not reported as heavy by only a small number of hash tables in the sketch; and 2) To reduce false positives we apply a second verifier sketch with 2-universal hash functions. In fact, we obtain analytical bounds on the false positives with this scheme.
- The IP-mangling scheme we design has good statistical properties that prevent attackers from subverting the heavy change detection system to create false alarms.

In addition, we implemented and evaluated our system with network traces obtained from two large edge routers with an OC-12 link or higher. The one day NU trace consists of 239M netflow records of 1.8TB total traffic. With a Pentium IV 2.4GHz PC, we record 1.6M packets per second. For inferring keys of even 1,000 heavy changes from two 5-minute traffic each recorded in a 3MB reversible sketch, our schemes find more than 99% of the heavy change keys with less than a 0.1% false positive rate within 13 seconds.

Both the analytical and experimental results show that we are able to achieve online traffic monitoring and accurate change/anomaly detection over massive data streams on high speed links, all in a manner that scales to large key space size. To the best of our knowledge, our system is the *first* to achieve these properties simultaneously.

In addition, as a sample application of reversible sketches, we briefly describe a sketch-based statistical flow-level intrusion detection and mitigation system (IDMS) that we designed and implemented (details are in a separate technical report [12]). We demonstrate that it can detect almost all SYN flooding and port scans (for most worm propagation) that can be found using complete flow-level logs, while with much less memory/space consumption and much faster monitoring and detection speed.

The rest of the paper is organized as follows. We give an overview of the data stream model and k-ary sketches in Section II. In Section III we discuss the algorithms for streaming data recording and in Section IV discuss those for heavy change detection. The application is briefly discussed in Section V. We evaluate our system in Section VI, survey

related work in Section VII, and finally conclude in Section VIII.

II. Overview

A. Data Stream Model and the k-ary Sketch

The Turnstile Model [13] is one of the most general data stream models. Let $I = \alpha_1, \alpha_2, \ldots,$ be an input stream that arrives sequentially, item by item. Each item $\alpha_i = (a_i, u_i)$ consists of a key $a_i \in [n]$, where $[n] = \{0, 1, \ldots, n-1\}$, and an update $u_i \in \mathbb{R}$. Each key $a \in [n]$ is associated with a time varying signal $U[a]$. Whenever an item (a_i, u_i) arrives, the signal $U[a_i]$ is incremented by u_i.

To efficiently keep accurate estimates of the signals $U[a]$, we use the k-ary sketch data structure. A k-ary sketch consists of H hash tables of size m (the k in the name k-ary sketch comes from the use of size k hash tables. However, in this paper we use m as the size of the hash tables, as is standard). The hash functions for each table are chosen independently at random from a class of 2-universal hash functions from $[n]$ to $[m]$. We store the data structure as an $H \times m$ table of registers $T[i][j]$ $(i \in [H], j \in [m])$. Denote the hash function for the i^{th} table by h_i. Given a data key and an update value, k-ary sketch supports the operation INSERT(a,u) which increments the count of bucket $h_i(a)$ by u for each hash table h_i. Let $D = \sum_{j \in [m]} T[0][j]$ be the sum of all updates to the sketch (the use of hash table 0 is an arbitrary choice as all hash tables sum to the same value). If an INSERT(a,u) operation is performed for each (key, update) pair in a data stream, then for any given key in a data stream, for each hash table the value $\frac{T[i][h_i(a)] - D/m}{1 - 1/m}$ constitutes an unbiased estimator for $U[a]$ [1]. A sketch can then provide a highly accurate estimate U_a^{est} for any key a, by taking the median of the H hash table estimates. See [1] or Theorem 2 for details on how to choose H and m to obtain quality estimates.

B. Change Detection

1) Absolute Change Detection: K-ary sketches can be used in conjunction with various forcasting models to perform sophisticated change detection as discussed in [1]. While all of our techniques in this paper are easily applicable to any of the forcast models in [1], for simplicity in this paper we focus on the simple model of change detection in which we break up the sequence of data items into two temporally adjacent chunks. We are interested in keys whose signals differ dramatically in size when taken over the first chunk versus the second chunk. In particular, for a given percentage ϕ, a key is a *heavy change key* if the difference in its signal exceeds ϕ percent of the total change over all keys. That is, for two input sets 1 and 2, if the signal for a key x is $U_1[x]$ over the first input and $U_2[x]$ over the second, then the difference signal for x is defined to be $D[x] = |U_1[x] - U_2[x]|$. The total difference is $D = \sum_{x \in [n]} D[x]$. A key x is then defined to be a heavy change key if and only if $D[x] \geq \phi \cdot D$. Note that this definition describes *absolute* change and does not characterize the potentially interesting set of keys with small signals that exhibit large change relative to their own size.

In our approach, to detect the set of heavy keys we create two k-ary sketches, one for each time interval, by updating them for each incoming packet. We then subtract the two sketches. Say S_1 and S_2 are the sketches recorded for the two consecutive time intervals. For detecting significant change in these two time periods, we obtain the difference sketch $S_d = |S_2 - S_1|$. The linearity property of sketches allows us to add or subtract a sketch to obtain estimates of the sum or difference of flows. Any key whose estimate value in S_d that exceeds the threshold $\phi \cdot D$ is denoted as a *suspect* heavy key in sketch S_d and offered as a proposed element of the set of heavy change keys.

2) Relative Change Detection: An alternate form of change detection is considered in [3]. In *relative* heavy change detection the change of a key is defined to be $D[x]_{rel} = \frac{U_1[x]}{U_2[x]}$. However, it is known that approximating the ratio of signals accurately requires a large amount of space [14]. The work in [3] thus limits itself to a form of *pseudo* relative change detection in which the exact values of all signals $U_1[x]$ are assumed to be known and only the signals $U_2[x]$ need to be estimated by updates over a data stream. Let $U_1 = \sum_{x \in [n]} \frac{1}{U_1[x]}$, $U_2 = \sum_{x \in [n]} U_2[x]$. For this limited problem, the following relative change estimation bounds for k-ary sketches can be shown.

Theorem 1: For a k-ary sketch which uses 2-universal hash functions, if $m = \frac{4}{\epsilon}$ and $H = 4 \log \frac{1}{\delta}$, then for all $x \in [n]$

$$D[x]_{rel} > \phi D + \epsilon U_1 U_2 \Rightarrow Pr[U_x^{est} < \phi \cdot D] < \delta$$

$$D[x]_{rel} < \phi D - \epsilon U_1 U_2 \Rightarrow Pr[U_x^{est} > \phi \cdot D] < \delta$$

Similar to Theorem 2, this bound suggests that our algorithms could be used to effectively solve the relative change problem as well. However, due to the limited motivation for pseudo relative change detection, we do no experiment with this problem.

TABLE I
Table of Notations

H	number of hash tables
$m = k$	number of buckets per hash table
n	size of key space
q	number of words keys are broken into
h_i	i^{th} hash function
$h_{i,1}, h_{i,2}, \ldots, h_{i,q}$	q modular hash functions that make up h_i
$\sigma_w(x)$	the w^{th} word of a q word integer x
$T[i][j]$	bucket j in hash table i
ϕ	percentage of total change required to be heavy
$h_{i,w}^{-1}$	an $m^{\frac{1}{q}} \times (\frac{n}{m})^{\frac{1}{q}}$ table of $\frac{1}{q} \log n$ bit words.
$h_{i,w}^{-1}[j][k]$	the k^{th} $n^{\frac{1}{q}}$ bit key in the reverse mapping of j for $h_{i,w}$
$h_{i,w}^{-1}[j]$	the set of all $x \in [n^{\frac{1}{q}}]$ s.t. $h_{i,w}(x) = j$
t'	number of heavy change keys
t	maximum number of heavy buckets per hash table
t_i	number of heavy buckets in hash table i
$t_{i,j}$	bucket index of the j^{th} heavy bucket in hash table i
r	number of hash tables a key can miss and still be considered heavy
I_w	set of modular keys occurring in heavy buckets in at least $H - r$ hash tables for the w^{th} word
$B_w(x)$	vector denoting for each hash table the set of heavy buckets modular key $x \in I_w$ occurs in

C. Problem Formulation

Instead of focusing directly on finding the set of keys that have heavy change, we instead attempt to find the set of keys

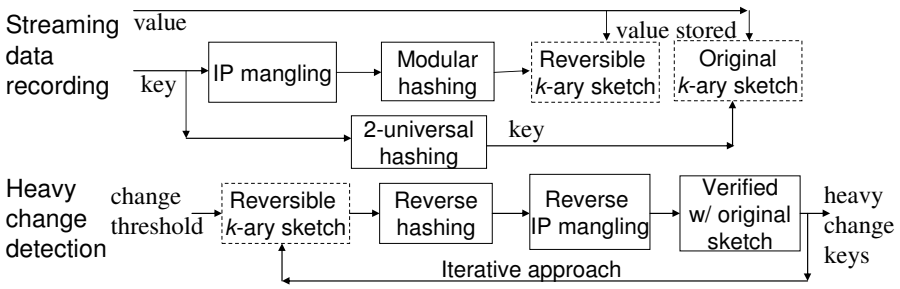


Fig. 1. Architecture of the reversible k-ary-sketch-based heavy change detection system for massive data streams.

denoted as suspects by a sketch. That is, our goal is to take a given sketch T with total traffic sum D, along with a threshold percentage ϕ, and output all the keys whose estimates in T exceed $\phi \cdot D$. We thus are trying to find the set of suspect keys for T.

To find this set, we can think of our input as a sketch T in which certain buckets in each hash table are marked as *heavy*. In particular, we denote the j^{th} bucket in hash table i as heavy if the value $\frac{T[i][j]-D/m}{1-1/m} \geq \phi D$. Thus, the j^{th} bucket in hash table i is heavy iff $T[i][j] \geq \phi(D-1/m)+D/m$. Thus, since the estimate for a sketch is the median of the estimates for each hash table, the goal is to output any key that hashes to a heavy bucket in more than $\lfloor \frac{H}{2} \rfloor$ of the H hash tables. If we let t be the maximum number of distinct heavy buckets over all hash tables, and generalize this situation to the case of mapping to heavy buckets in at least $H-r$ of the hash tables where r is the number of hash tables a key can miss and still be considered heavy, we get the following problem.

The Reverse Sketch Problem
Input:

- Integers $t \geq 1$, $r < \frac{H}{2}$;
- A sketch T with hash functions $\{h_i\}_{i=0}^{H-1}$ from $[n]$ to $[m]$;
- For each hash table i a set of at most t *heavy* buckets $R_i \subseteq [m]$;

Output: All $x \in [n]$ such that $h_i(x) \in R_i$ for $H-r$ or more values $i \in [H]$.

In section IV we show how to efficiently solve this problem.

D. Bounding False Positives

Since we are detecting suspect keys for a sketch rather than directly detecting heavy change keys, we discuss how accurately the set of suspect keys approximates the set of heavy change keys. Let $S_d = |S_2 - S_1|$ be a difference sketch over two data streams. For each key $x \in [n]$ denote the value of the difference of the two signals for x by $D[x] = |U_2[x] - U_1[x]|$. Denote the total difference by $D = \sum_{x \in [n]} D[x]$. The following theorem relates the size of the sketch (in terms of m and H) with the probability of a key being incorrectly categorized as a heavy change key or not.

Theorem 2: For a k-ary sketch which uses 2-universal hash functions, if $m = \frac{8}{\epsilon}$ and $H = 4 \log \frac{1}{\delta}$, then for all $x \in [n]$

$$D[x] > (\phi + \epsilon) \cdot D \Rightarrow Pr[U_x^{est} < \phi \cdot D] < \delta$$
$$D[x] < (\phi - \epsilon) \cdot D \Rightarrow Pr[U_x^{est} > \phi \cdot D] < \delta$$

Intuitively this theorem states that if a key is an ϵ-*approximate* heavy change key, then it will be a suspect with probability at least $1-\delta$, and if it is an ϵ-approximate non-heavy key, it will not be a suspect with probability at least $1-\delta$. We can thus make the set of suspect keys for a sketch an appropriately good approximation for the set of heavy change keys by choosing large enough values for m and H. We omit the proof of this theorem in the interest of space, but refer the reader to [3] in which a similar theorem is proven.

As we discuss in Section III-A, our reversible k-ary sketch does not have 2-universality. However, we use a second non-reversible k-ary sketch with 2-universal functions to act as a verifier for any suspect keys reported. This gives our algorithm the analytical limitation on false positives of theorem 2. As an optimization we can thus leave the reduction of false positives to the verifier and simply try to output as many suspect keys as is feasible. For example, to detect the heavy change keys with respect to a given percentage ϕ, we could detect the set of suspect keys for the initial sketch with respect to $\phi - \alpha$, for some percentage α, and then verify those suspects with the second sketch with respect to ϕ. However, we note that even without this optimization (setting $\alpha = 0$) we obtain very high true-positive percentages in our simulations.

E. Architecture

Our change detection system has two parts (Fig. 1): streaming data recording and heavy change detection as discussed below.

III. Streaming Data Recording

The first phase of the change detection process is passing over each data item in the stream and updating the summary data structure. The update procedure for a k-ary sketch is very efficient. However, with standard hashing techniques the detection phase of change detection cannot be performed efficiently. To overcome this we modify the update for the k-ary sketch by introducing *modular hashing* and *IP mangling* techniques.

A. Modular hashing

Modular hashing is illustrated in Figure 2. Instead of hashing the entire key in $[n]$ directly to a bucket in $[m]$, we partition the key into q words, each word of size $\frac{1}{q} \log n$ bits. Each word is then hashed separately with different hash functions which map from space $[n^{\frac{1}{q}}]$ to $[m^{\frac{1}{q}}]$. For example, in Figure 2, a 32-bit IP address is partitioned into $q = 4$ words, each of 8 bits. Four independent hash functions are then chosen which map from space $[2^8]$ to $[2^3]$. The results of

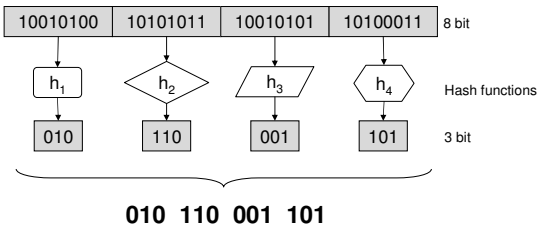


Fig. 2. Illustration of modular hashing.

each of the hash functions are then concatenated to form the final hash. In our example, the final hash value would consist of 12 bits, deriving each of its 3 bits from the separate hash functions $h_{i,1}, h_{i,2}, h_{i,3}, h_{i,4}$. If it requires constant time to hash a value, modular hashing increases the update operations from $O(H)$ to $O(q \cdot H)$. On the other hand, no extra memory access is needed. Furthermore, in section IV we will discuss how modular hashing allows us to efficiently perform change detection. However, an important issue with modular hashing is the quality of the hashing scheme. The probabilistic estimate guarantees for k-ary sketch assume 2-universal hash functions, which can map the input keys uniformly over the buckets. In network traffic streams, we notice strong spatial localities in the IP addresses, *i.e.*, many simultaneous flows only vary in the last few bits of their source/destination IP addresses, and share the same prefixes. With the basic modular hashing, the collision probability of such addresses are significantly increased.

For example, consider a set of IP addresses $129.105.56.*$ that share the first 3 octets. Modular hashing always maps the first 3 octets to the same hash values. Thus, assuming our small hash functions are completely random, all distinct IP addresses with these octets will be uniformly mapped to 2^3 buckets, resulting in a lot of collisions. This observation is further confirmed when we apply modular hashing to the network traces used for evaluation (see Section VI). The distribution of the number of keys per bucket is highly skewed, with most of the IP addresses going to a few buckets (Figure 3). This significantly disrupts the estimation accuracy of the reversible k-ary sketch. To overcome this problem, we introduce the technique of *IP mangling*.

B. Attack-resilient IP Mangling

In IP mangling we attempt to artificially randomize the input data in an attempt to destroy any correlation or spatial locality in the input data. The objective is to obtain a completely random set of keys, and this process should be still reversible.

The general framework for the technique is to use a bijective function from key space $[n]$ to $[n]$. For an input data set consisting of a set of distinct keys $\{x_i\}$, we map each x_i to $f(x_i)$. We then use our algorithm to compute the set of proposed heavy change keys $C = \{y_1, y_2, \ldots, y_c\}$ on the input set $\{f(x_i)\}$. We then use f^{-1} to output $\{f^{-1}(y_1), f^{-1}(y_2), \ldots, f^{-1}(y_c)\}$, the set of proposed heavy change keys under the original set of input keys. Essentially, we transform the input set to a mangled set and perform all our operations on this set. The output is then transformed back to the original input keys.

1) Attack-resilient Scheme: In [2] the function $f(x) = a \cdot x \pmod n$ is proposed where a is an odd integer chosen uniformly at random. This function can be computed quickly (no taking mod of a prime) and is effective for hierarchical key spaces such as IP addresses where it is natural to assume no traffic correlation exists among any two keys that have different (non-empty) prefixes. However, this is not a safe assumption in general. And even for IP addresses, it is plausible that an attacker could antagonistically cause a non-heavy-change IP address to be reported as a false positive by creating large traffic changes for an IP address that has a similar suffix to the target - also known as *behavior aliasing*. To prevent such attacks, we need the mapping of any pair of distinct keys to be independent of the choice of the two keys. That is, we want a universal mapping.

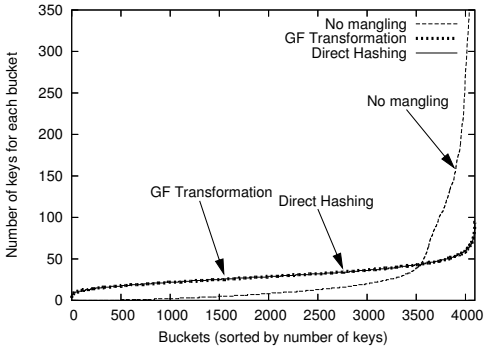


Fig. 3. Distribution of number of keys for each bucket under three hashing methods. Note that the plots for direct hashing and the GF transformation are essentially identical.

We propose the following universal hashing scheme based on simple arithmetic operations on a Galois Extension Field [15] $\mathbb{GF}(2^\ell)$, where $\ell = \log_2 n$. More specifically, we choose a and b from $\{1, 2, \cdots, 2^\ell - 1\}$ uniformly at random, and then define $f(x) \equiv a \otimes x \oplus b$, where '$\otimes$' is the multiplication operation defined on $\mathbb{GF}(2^\ell)$ and '$\oplus$' is the bit-wise XOR operation. We refer to this as the Galois Field (GF) transformation. By precomputing a^{-1} on $\mathbb{GF}(2^\ell)$, we can easily reverse a mangled key y using $f^{-1}(y) = a^{-1} \otimes (y \oplus b)$.

The direct computation of $a \otimes x$ can be very expensive, as it would require multiplying two polynomials (of degree $\ell - 1$) modulo an irreducible polynomial (of degree ℓ) on a Galois Field $\mathbb{GF}(2)$. In our implementation, we use tabulation to speed up the computation of $a \otimes x$. The basic idea is to divide input keys into shorter characters. Then, by precomputing the product of a and each character we can translate the computation of $a \otimes x$ into a small number of table lookups. For example, with 8-bit characters, a given 32-bit key x can be divided into four characters: $x = \overline{x_3 x_2 x_1 x_0}$. According to the finite field arithmetic, we have $a \otimes x = a \otimes \overline{x_3 x_2 x_1 x_0} = \bigoplus_{i=0}^{3} a \otimes (x_i \ll 8\ i)$, where '$\oplus$' is the bit-wise XOR operation, and $\ll$ is the shift operation. Therefore, by precomputing 4 tables $t_i[0..255]$, where $t_i[y] = a \otimes (y \ll 8\ i)$ ($\forall i = 0..3$, $\forall y = 0..255$), we can efficiently compute $a \otimes x$ using four table lookups: $a \otimes x = t_3[x_3] \oplus t_2[x_2] \oplus t_1[x_1] \oplus t_0[x_0]$.

We can apply the same approach to compute f and f^{-1} (with separate lookup tables). Depending on the amount of resource available, we can use different character lengths. For

our hardware implementation, we use 8-bit characters so that the tables are small enough to fit into fast memory ($2^8 \times 4 \times 4Bytes = 4KB$ for 32-bit IP addresses). Note that only IP mangling needs extra memory and extra memory lookup as modular hashing can be implemented efficiently without table lookup. For our software implementation, we use 16-bit characters, which is faster than 8-bit characters due to fewer table lookups.

In practice this mangling scheme effectively resolves the highly skewed distribution caused by the modular hash functions. Using the source IP address of each flow as the key, we compare the hashing distribution of the following three hashing methods with the real network flow traces: 1) modular hashing with no IP mangling, 2) modular hashing with the GF transformation for IP mangling, and 3) direct hashing (a completely random hash function). Figure 3 shows the distribution of the number of keys per bucket for each hashing scheme. We observe that the key distribution of modular hashing with the GF transformation is essentially the same as that of direct hashing. The distribution for modular hashing without IP mangling is highly skewed. Thus IP mangling is very effective in randomizing the input keys and removing hierarchical correlations among the keys.

In addition, our scheme is resilient to behavior aliasing attacks because attackers cannot create collisions in the reversible sketch buckets to make up false positive heavy changes. Any distinct pair of keys will be mapped completely randomly to two buckets for *each* hash table.

IV. Reverse Hashing

We now discuss how modular hashing permits the efficient execution of the detection phase of the change detection process. To provide an initial intuition, we start with the simple (but somewhat unrealistic) scenario in which we have a sketch taken over a data stream that contains exactly one heavy bucket in each hash table. Our goal is to output any key value that hashes to the heavy bucket for most of the hash tables. For simplicity, let's assume we want to find all keys that hit the heavy bucket in every hash table. We thus want to solve the reverse sketch problem for $t = 1$ and $r = 0$.

To find this set of culprit keys, consider for each hash table the set A_i consisting of all keys in $[n]$ that hash to the heavy bucket in the i^{th} hash table. We thus want to find $\bigcap_{i=0}^{H-1} A_i$. The problem is that each set A_i is of expected size $\frac{n}{m}$, and is thus quite large. However, if we are using modular hashing, we can implicity represent each set A_i by the cross product of q *modular* reverse mapping sets $A_{i,1} \times A_{i,2} \times \cdots A_{i,q}$ determined by the corresponding modular hash functions $h_{i,w}$. The pairwise intersection of any two reverse mapping sets is then $A_i \bigcap A_j = A_{i,1} \bigcap A_{j,1} \times A_{i,2} \bigcap A_{j,2} \times \cdots \times A_{i,q} \bigcap A_{j,q}$. We can thus determine the desired H-wise intersection by dealing with only the smaller *modular* reverse mapping sets of size $(\frac{n}{m})^{\frac{1}{q}}$. This is the basic intuition for why modular hashing might improve the efficiency of performing reverse hashing and constitutes the approach used in [16].

A. Simple Extension Doesn't Work

Extending the intuitions for how to reverse hash for the case where $t = 1$ to the case where $t \geq 1$ is not trivial. Consider the simple case of $t = 2$, as shown in Figure 4. There are now $t^H = 2^H$ possible ways to take the H-wise intersections discussed for the $t = 1$ case. One possible heuristic is to take the union of the possible keys of all heavy change buckets for each hash table and then take the intersections of these unions. However, this can lead to a huge number of keys output that do not fulfill the requirement of our problem. In fact, we have shown (proof omitted) that for arbitrary modular hash functions that evenly distribute $\frac{n}{m}$ keys to each bucket in each hash table, there exist extreme cases such that the Reverse Sketch Problem cannot be solved for $t \geq 2$ in polynomial time in both q and H in general, even when the size of the output is $O(1)$ unless $P = NP$. We thus are left to hope for an algorithm that can take advantage of the random modular hash functions described in Section III-A to solve the reverse sketch problem efficiently with high probability. The remainder of this section describes our general case algorithm for resolving this problem.

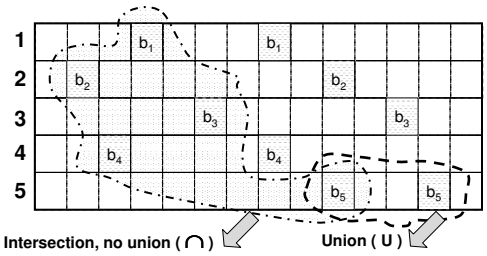


Fig. 4. For the case of $t = 2$, various possibilities exist for taking the intersection of each bucket's potential keys

B. Notation for the General Algorithm

We now introduce our general method of reverse hashing for the more realistic scenarios where there are multiple heavy buckets in each hash table and we allow for the possibility that a heavy change key can miss a heavy bucket in a few hash tables. That is, we present an algorithm to solve the reverse sketch problem for any t and r that is assured to obtain the correct solution with a polynomial run time in q and H with very high probability. To describe this algorithm, we define the following notation.

Let the i^{th} hash table contain t_i heavy buckets. Let t be the value of the largest t_i. For each of the H hash tables h_i, assign an arbitrary indexing of the t_i heavy buckets and let $t_{i,j} \in [m]$ be the index in hash table i of heavy bucket number j. Also define $\sigma_w(x)$ to be the w^{th} word of a q word integer x. For example, if the j^{th} heavy bucket in hash table i is $t_{i,j} = 5.3.0.2$ for $q = 4$, then $\sigma_2(t_{i,j}) = 3$.

For each $i \in [H]$ and word w, denote the reverse mapping set of each modular hash function $h_{i,w}$ by the $m^{\frac{1}{q}} \times (\frac{n}{m})^{\frac{1}{q}}$ table $h_{i,w}^{-1}$ of $\frac{1}{q} \log n$ bit words. That is, let $h_{i,w}^{-1}[j][k]$ denote the k^{th} $n^{\frac{1}{q}}$ bit key in the reverse mapping of j for $h_{i,w}$. Further, let $h_{i,w}^{-1}[j] = \{x \in [n^{\frac{1}{q}}] \mid h_{i,w}(x) = j\}$.

Let $I_w = \{x \mid x \in \bigcup_{j=0}^{t_i-1} h_{i,w}^{-1}[\sigma_w(t_{i,j})]$ for at least $H - r$ values $i \in [H]\}$. That is, I_w is the set of all $x \in [n^{\frac{1}{q}}]$ such that x is in the reverse mapping for $h_{i,w}$ for some heavy bucket in at least $H - r$ of the H hash tables. We occasionally refer to

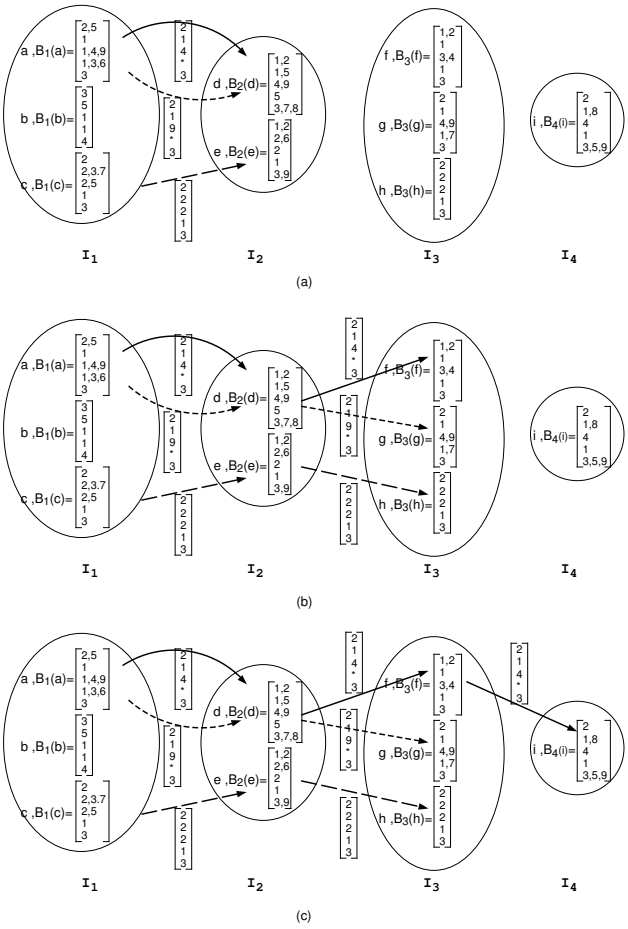


Fig. 5. Given the q sets I_w and bucket index matrices B_w we can compute the sets A_w incrementally. The set A_2 containing $(\langle a, d\rangle, \langle 2, 1, 4, *, 3\rangle)$, $(\langle a, d\rangle, \langle 2, 1, 9, *, 3\rangle)$, and $(\langle c, e\rangle, \langle 2, 2, 2, 1, 3\rangle)$ is depicted in (a). From this we determine the set A_3 containing $(\langle a, d, f\rangle, \langle 2, 1, 4, *, 3\rangle)$, $(\langle a, d, g\rangle, \langle 2, 1, 9, *, 3\rangle)$, and $(\langle c, e, h\rangle, \langle 2, 2, 2, 1, 3\rangle)$ shown in (b). Finally we compute A_4 containing $(\langle a, d, f, i\rangle, \langle 2, 1, 4, *, 3\rangle)$ shown in (c).

this set as the *intersected modular potentials* for word w. For instance, in Figure 5, I_1 has three elements and I_2 has two.

For each word we also define the mapping B_w which specifies for any $x \in I_w$ exactly which heavy buckets x occurs in for each hash table. In detail, $B_w(x) = \langle L_w[0][x], L_w[1][x], \ldots, L_w[H-1][x]\rangle$ where $L_w[i][x] = \{j \in [t] \mid x \in h^{-1}_{i,w}[\sigma_w(t_{i,j})]\} \bigcup \{*\}$. That is, $L_w[i][x]$ denotes the collection of indices in $[t]$ such that x is in the modular bucket potential set for the heavy bucket corresponding to the given index. The special character * is included so that no intersection of sets L_w yields an empty set. For example, $B_w(129) = \langle\{1, 3, 8\}, \{5\}, \{2, 4\}, \{9\}, \{3, 2\}\rangle$ means that the reverse mapping of the 1^{st}, 3^{rd}, and 8^{th} heavy bucket under $h_{0,w}$ all contain the modular key 129.

We can think of each vector $B_w(x)$ as a set of all H dimensional vectors such that the i^{th} entry is an element of $L_w[i][x]$. For example, $B_3(23) = \langle\{1, 3\}, \{16\}, \{*\}, \{9\}, \{2\}\rangle$ is indeed a set of two vectors: $\langle\{1\}, \{16\}, \{*\}, \{9\}, \{2\}\rangle$ and $\langle\{3\}, \{16\}, \{*\}, \{9\}, \{2\}\rangle$. We refer to $B_w(x)$ as the *bucket index matrix* for x, and a decomposed vector in a set $B_w(x)$ as a *bucket index vector* for x. We note that although the size of the bucket index vector set is exponential in H, the bucket index matrix representation is only polynomial in size and permits the operation of intersection to be performed in polynomial time. Such a set like $B_1(a)$ can be viewed as a *node* in Figure 5.

Define the r *intersection* of two such sets to be $B \bigcap^r C = \{v \in B \bigcap C \mid v$ *has at most* r *of its* H *entries equal to* $*$ $\}$. For example, $B_w(x) \bigcap^r B_{w+1}(y)$ represents all of the different ways to choose a single heavy bucket from each of at least $H - r$ of the hash tables such that each chosen bucket contains x in it's reverse mapping for the w^{th} word and y for the $w{+}1^{th}$ word. For instance, in Figure 5, $B_1(a) \bigcap^r B_2(d) = \langle\{2\}, \{1\}, \{4\}, \{*\}, \{3\}\rangle$, which is denoted as a *link* in the figure. Note there is no such link between $B_1(a)$ and $B_2(e)$. Intuitively, the $a.d$ sequence can be part of a heavy change key because these keys share common heavy buckets for at least $H - r$ hash tables. In addition, it is clear that a key $x \in [n]$ is a suspect key for the sketch if and only if $\bigcap^r_{w=1\ldots q} B_w(x_w) \neq \emptyset$.

Finally, we define the sets A_w which we compute in our algorithm to find the suspect keys. Let $A_1 = \{(\langle x_1\rangle, v) \mid x_1 \in I_1$ and $v \in B_1(x_1)\}$. Recursively define $A_{w+1} = \{(\langle x_1, x_2, \ldots, x_{w+1}\rangle, v) \mid (\langle x_1, x_2, \ldots, x_w\rangle, v) \in A_w$ and $v \in B_{w+1}(x_{w+1})\}$. Take Figure 5 for example. Here A_4 contains $\langle a, d, f, i\rangle, \langle 2, 1, 4, *, 3\rangle$ which is the suspect key. Each element of A_w can be denoted as a *path* in Figure 5. The following lemma tells us that it is sufficient to compute A_q to solve the reverse sketch problem.

Lemma 1: A key $x = x_1.x_2.\cdots.x_q \in [n]$ is a suspect key if and only if $(\langle x_1, x_2, \cdots, x_q\rangle, v) \in A_q$ for some vector v.

C. Algorithm

To solve the reverse sketch problem we first compute the q sets I_w and bucket index matrices B_w. From these we iteratively create each A_w starting from some base A_c for any c where $1 \leq c \leq q$ up until we have A_q. We then output the set of heavy change keys via lemma (1). Intuitively, we start with nodes as in Figure 5, I_1 is essentially A_1. The links between I_1 and I_2 give A_2, then the link pairs between (I_1 I_2) and (I_2 I_3) give A_3, etc.

The choice of the base case A_c affects the performance of the algorithm. The size of the set A_1 is likely to be exponentially large in H. However, with good random hashing, the size of A_w for $w \geq 2$ will be only polynomial in H, q, and t with high probability with the detailed algorithm and analysis below. Note we must choose a fairly small value c to start with because the complexity of computing the base case grows exponentially in c.

REVERSE_HASH(r)

1 For each $w = 1$ to q, set $(I_w, B_w) =$ MODULAR_POTENTIALS(w, r).
2 Initialize $A_2 = \emptyset$. For each $x \in I_1$, $y \in I_2$, and corresponding $v \in B_1(x) \bigcap^r B_2(y)$, insert $(\langle x, y\rangle, v)$ into A_2.
3 For any given A_w set $A_{w+1} =$ Extend(A_w, I_{w+1}, B_{w+1}).
4 Output all $x_1.x_2.\cdots.x_q \in [n]$ s.t. $(\langle x_1, \ldots, x_q\rangle, v) \in A_q$ for some v.

MODULAR_POTENTIALS(w, r)

1 Create an $H \times n^{\frac{1}{q}}$ table of sets L initialized to all contain the special character *. Create a size $[n^{\frac{1}{q}}]$ array of counters *hits* initialized to all zeros.
2 For each $i \in [H]$, $j \in [t]$, and $k \in [(\frac{n}{m})^{\frac{1}{q}}]$ insert $h_{i,w}^{-1}[\sigma_w(t_{i,j})][k]$ into $L[i][x]$. If $L[i][x]$ was empty, increment $hits[x]$.
3 For each $x \in [n^{\frac{1}{q}}]$ s.t. $hits[x] \geq H - r$, insert x into I_w and set $B_w(x) = \langle L[0][x], L[1][x], \ldots, L[H-1][x] \rangle$.
4 Output (I_w, B_w).

EXTEND(A_w, I_{w+1}, B_{w+1})
1 Initialize $A_{w+1} = \emptyset$.
2 For each $y \in I_{w+1}$, $(\langle x_1, \ldots, x_w \rangle, v) \in A_w$, determine if $v \bigcap^r B_{w+1}(y) \neq$ null. If so, Insert $(\langle x_1, \ldots, x_w, y \rangle$, $v \bigcap^r B_{w+1}(y))$ into A_{w+1}.
3 Output A_{w+1}.

D. Complexity Analysis

Lemma 2: The number of elements in each set I_w is at most $\frac{H}{H-r} \cdot t \cdot (\frac{n}{m})^{\frac{1}{q}}$.

Proof: Each element x in I_w must occur in the modular potential set for some bucket in at least $H - r$ of the H hash tables. Thus at least $|I_w| \cdot (H - r)$ of the elements in the multiset of modular potentials must be in I_w. Since the number of elements in the multiset of modular potentials is at most $H \cdot t \cdot (\frac{n}{m})^{\frac{1}{q}}$ we get the following inequality.
$|I_w| \cdot (H - r) \leq H \cdot t \cdot (\frac{n}{m})^{\frac{1}{q}} \Longrightarrow |I_w| \leq \frac{H}{H-r} \cdot t \cdot (\frac{n}{m})^{\frac{1}{q}}$ ■

Next, we will show that the size of A_w will be only polynomial in H, q and t.

Lemma 3: With proper m and t, the number of bucket index vectors in A_2 is $O(n^{2/q})$ with high probability.

In the interest of space we refer the reader to the full technical report for the details of this proof [16].

Given Lemma 3, the more heavy buckets we have to consider, the bigger m must be, and the more memory is needed. Take the 32-bit IP address key as an example. In practice, $t \leq m^{2/q}$ works well. When $q = 4$ and $t \leq 64$, we need $m = 2^{12}$. For the same q, when $t \leq 256$, we need $m = 2^{16}$, and when $t \leq 1024$, we need $m = 2^{20}$. This may look prohibitive. However, with the iterative approach in Section IV-F, we are able to detect many more changes with small m. For example, we are able to detect more than 1000 changes accurately with $m = 2^{16}$ (1.5MB memory needed) as evidenced in the evaluations (Section VI). Since we normally only consider at most the top 50 to a few hundred heavy changes, we can have $m = 2^{12}$ with memory less than 100KB.

Lemma 4: With proper choices of H, r, and m, the expected number of bucket index vectors in A_{w+1} is less than that of A_w for $w \geq 2$.
That is, the expected number of link sequences with length $x + 1$ is less than the number of link sequences with length x when $x \geq 2$.

Proof: For any bucket index vector $v \in A_w$, for any word $x \in [n^{1/q}]$ for word $w+1$, the probability for x to be in the same ith ($i \in [H]$) bucket is $\frac{1}{m^{1/q}}$. Thus the probability for $B(x) \bigcap^r v$ to be not null is at most $C_{H-r}^H \times \frac{1}{m^{(H-r)/q}}$. Given there are $n^{1/q}$ possible words for word $w+1$, the probability for any v to be extensible to A_{w+1} is $C_{H-r}^H \times \frac{1}{m^{(H-r)/q}} \times n^{1/q}$. With proper H, r and m for any n, we can easily have such probability to be smaller than 1. Then the number of bucket index vectors in A_{w+1} is less than that of A_w. ■

Given the lemmas above, `MODULAR_POTENTIALS` and step 2 of `REVERSE_HASH` run in time $O(n^{2/q})$. The running time of `EXTEND` is $O(n^{3/q})$. So the total running time is $O((q-2) \cdot n^{3/q})$.

E. Asymptotic Parameter Choices

To make our scheme run efficiently and maintain accuracy for large values of n, we need to carefully choose the parameters m, H, and q as functions of n. Our data structures and algorithms for the streaming update phase use space and time polynomial in H, q, and m, while for the change detection phase they use space and time polynomial in H, q, m, and $n^{\frac{1}{q}}$. Thus, to maintain scalability, we must choose our parameters such that all of these values are sufficiently smaller than n. Further, to maintain accuracy and a small sketch size, we need to make sure the following constraints are satisfied.

First, to limit the number of collisions in the sketch, for any choice of a single bucket from each hash table, we require that the expected number of keys to hash to that sequence be bounded by some small parameter ϵ, $\frac{n}{m^H} < \epsilon$. Second, the modular bucket size must be bounded below by a constant, $m^{\frac{1}{q}} > c$. Third, we require that the total sketch size mH be bounded by a polynomial in $\log n$. Given these constraints we are able to maintain the following parameter bounds. For an extended discussion motivating these parameter choices please see the full technical report [16].

$$q = \log\log n \qquad m = (\log n)^{\Theta(1)}$$
$$n^{\frac{1}{q}} = n^{\frac{1}{\log\log n}} \qquad H = O(\frac{\log n}{\log\log n})$$

F. Iterative Detection

From our discussion in Section IV-D we have that our detection algorithm can only effectively handle t of size at most $m^{\frac{2}{q}}$. With our discussion in Section IV-E this is only a constant. To handle larger t, consider the following heuristic. Suppose we can comfortably handle at most c heavy buckets per hash table. If a given ϕ percentage results in $t > c$ buckets in one or more tables, sort all heavy buckets in each hash table according to size. Next, solve the reverse sketch problem with respect to only the largest c heavy buckets from each table. For each key output, obtain an estimate from a second k-ary sketch independent of the first. Update each key in the output by the negative of the estimate provided by the second sketch. Having done this, once again choose the largest c buckets from each hash table and repeat. Continue until there are no heavy buckets left.

One issue with this approach is that an early false positive (a key output that is not a heavy change key) will cause large numbers of false negatives since the (incorrect) decrement of the buckets for the false positive will potentially cause many false negatives in successive iterations. To help reduce this we can use the second sketch as a verifier for any output keys to reduce the possibility of a false positive in each iteration.

G. Comparison with the Deltoids Approach

The most related work to ours is the recently proposed *deltoids* approach for heavy change detection [3]. Though

TABLE II

A COMPARISON BETWEEN THE REVERSIBLE SKETCH METHOD AND THE DELTOIDS APPROACH. HERE t' DENOTES THE NUMBER OF HEAVY CHANGE KEYS IN THE INPUT STREAM. NOTE THAT IN EXPECTATION $t \geq t'$.

	Update			Detection	
	memory	memory accesses	operations	memory	operations
Reversible Sketch	$\Theta(\frac{(\log n)^{\Theta(1)}}{\log\log n})$	$\Theta(\frac{\log n}{\log\log n})$	$\Theta(\log n)$	$\Theta(n^{\frac{1}{\log\log n}} \cdot \log\log n)$	$O(n^{\frac{3}{\log\log n}} \cdot \log\log n \cdot t)$
Deltoids	$\Theta(\log n \cdot t')$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n \cdot t')$	$O(\log n \cdot t')$

developed independently of k-ary sketch, deltoid essentially expands k-ary sketch with multiple counters for each bucket in the hash tables. The number of counters is logarithmic to the key space size (*e.g.*, 32 for IP addresses), so that for every (key, value) entry, instead of adding the value to one counter in each hash table, it is added to multiple counters (32 for IP addresses and 64 for IP address pairs) in each hash table. This significantly increases the necessary amount of fast memory and number of memory accesses per packet, and is not scalable to large key space size such as 2^{104} discussed in Section I. Thus, it violates all the aforementioned performance constraints in Section I.

The advantage of the deltoids approach is that it is more efficient in the detection phase, with run time and space usage only logarithmic in the key space n. While our method does not achieve this, its run time and space usage is significantly smaller than the key space n. And since this phase of change detection only needs to be done periodically in the order of at most seconds, our detection works well for key sizes of practical interest. We summarize the asymptotic efficiencies of the two approaches in Table II, but omit details of the derivations in the interest of space. Note that the reversible sketch data structure offers an improvement over the deltoids approach in the number of memory accesses per update, as well as the needed size of the data structure when there are many heavy buckets (changes). Together this yields a significant improvement in achievable update speed.

V. APPLICATIONS

A. General Framework

The key feature of reversible sketches is to support aggregate queries over multiple data streams, *i.e.*, to find the top heavy hitters and their keys from the linear combination of multiple data streams for temporal and/or spatial aggregation. Many statistical approaches, such as Time Series Analysis (TSA), need this functionality for anomaly/trend detection. Take TSA as an example. In the context of network applications, there are often tens of millions of network time series and it is very hard, if not impossible, to apply the standard techniques on a per time series basis. Reversible sketches help solve this problem. Moreover, in today's networks, asymmetric routing, multi-homing, and load balancing are very common and many enterprises have more than one upstream or downstream link. For example, it is quite impossible to detect port scans or SYN flooding based on {SYN, SYN/ACK} or {SYN, FIN} pairs on a *single* router if the SYN, SYN/ACK and FIN for a particular flow can travel different routers or links. Again, the linearity of reversible sketches enables traffic aggregation over multiple routers to facilitate such detection.

B. Intrusion Detection and Mitigation on High-speed Networks

Global-scale attacks like viruses and worms are increasing in frequency, severity and sophistication, making it critical to detect outbursts at routers/gateways instead of end hosts. With reversible sketches, we have built a novel, high-speed statistical flow-level intrusion detection and mitigation system (IDMS) for TCP SYN flooding and port scan detection. In contrast to existing intrusion detection systems, the IDMS 1) is scalable to flow-level detection on high-speed networks (such as OC192); 2) is DoS resilient; 3) enables aggregate detection over multiple routers/gateways. We use three different reversible sketches to detect SYN flooding and the two most popular port scans: horizontal scans (for most worm propagation) and vertical scans (for attacking specific target machines). Reversible sketches reveal the IP addresses and ports that are closely related to the attacks. Appropriate counter-measures can then be applied. Take port scans and point-to-point SYN flooding for example. We can use ingress filters to block the traffic from the attacker IP. The evaluation based on router traffic as described in Section VI-B demonstrates that the reversible sketch based IDMS significantly outperforms existing approaches like Threshold Random Walk (TRW) [17], TRW with approximate caches [18], and Change-Point Monitoring [19], [20]. For more details, please refer to [12].

VI. IMPLEMENTATION AND EVALUATION

In this section, we first discuss the implementation and evaluation of streaming data recording in hardware. We then introduce the methodology and simulation results for heavy change detection.

A. Hardware Traffic Recording Achieves 16Gbps

The Annapolis WILDSTAR Board is used to implement the original and reversible k-ary sketch. This platform consists of three Xilinx Virtex 2000E FPGA chips [21], each with 2.5M gates contained within 9600 Configurable Logic Blocks (CLBs) interconnected via a cross-bar along with memory modules. This development board is hosted by a SUN Ultra-10 workstation. The unit is implemented using the Synplify Pro 7.2. tool [22]. Such FPGA boards cost about $1000.

The sketch hardware consists of H hash units, each of which addresses a single m-element array. For almost all configurations, delay is the bottleneck. Therefore, we have optimized it using excessive pipelining. The resulting maximum throughputs for 40-byte-packet streams for $H = 5$ are: For the original k-ary sketch, we achieve a high bandwidth of over 22 Gbps. For the reversible sketch with modular hashing we archive 19.3Gbps. Even for the reversible sketch with IP mangling and modular hashing, we achieve 16.2 Gbps.

B. Software Simulation Methodology

1) Network Traffic Traces: In this section we evaluate our schemes with *Netflow* traffic traces collected from two sources as shown in Table III.

TABLE III
EVALUATION DATA SETS

Collection Location	A large US ISP	Northwestern Univ.
# of Netflow records	330M	19M
peak packet rate	86K/sec	79K/sec
avg. packet rate	63K/sec	37K/sec

In both cases, the trace is divided into 5-minute intervals. For ISP data the traffic for each interval is about 6GB. The distribution of the heavy change traffic volumes (in Bytes) over 5 minutes for these two traces is shown in Figure 6. The y-axis is in logarithmic scale. Though having different traffic volume scales, the heavy changes of both traces follow heavy-tail distributions. In the interest of space, we focus on the ISP data. Results are the same for the Northwestern traces.

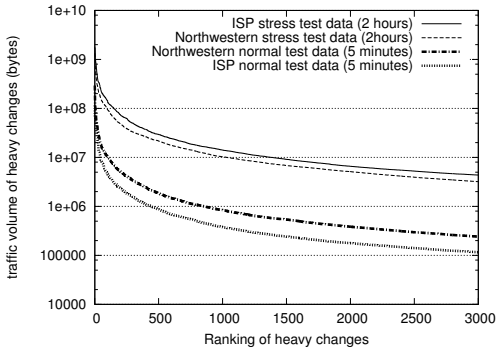


Fig. 6. The distribution of the top heavy changes for both data sets

2) Experimental Parameters: In this section, we present the values of parameters that we used in our experiments, and justify their choices.

The cost of sketch updating is dominated by the number of hash tables, so we choose small values for H. Meanwhile, H improves the accuracy by making the probability of hitting extreme estimates exponentially small [1]. We applied the "grid search" method in [1] to evaluate the impact on the accuracy of estimation with respect to cost, and obtained similar results as those for the original sketches. That is, it makes little difference to increase H much beyond 5. As a result, we choose H to be 5 and 6.

Given H, we also need to choose r. As in Section II-C, our goal is to output any key that hashes to a heavy bucket in more than $\lfloor \frac{H}{2} \rfloor$ of the H hash tables. Thus, we consider $r < \lfloor \frac{H}{2} \rfloor$ and the values $H = 5$, $r = 1$; and $H = 6$, $r = 1$ or 2.

Another important parameter is m, the number of buckets in each hash table. The lower bound for providing a reasonable degree of error threshold is found to be $m = 1024$ for normal sketches [1], which is also applicable to reversible sketches. Given that the keys are usually IP addresses (32 bits, $q = 4$) or IP address pairs (64 bits, $q = 8$), we want $m = 2^{xq}$ for an integer x. Thus, m should be at least 2^{12}.

We also want to use a small amount of memory so that the entire data structure can fit in fast SRAM. The total memory for update recording is only $2 \times \langle number\ of\ tables(H) \rangle \times \langle number\ of\ bins(m) \rangle \times 4bytes/bucket$. This includes a *reversible* k-ary sketch and an *original* k-ary sketch. In addition to the two settings for H, we experiment with two choices for m: 2^{12} and 2^{16}. Thus, the largest memory consumption is 3MB for $m = 2^{16}$ and $H = 6$, while the smallest one is 160KB for $m = 2^{12}$ and $H = 5$.

We further compare it with the state-of-the-art *deltoids* approach (see Section IV-G), with the *deltoids* software provided by its authors. To obtain a fair comparison we allot equal memory to each method, *i.e.*, the memory consumption of the reversible sketch and the verifying sketch equals that of the $deltoids$.

3) Evaluation Metrics: Our metrics include *accuracy* (in terms of the real positive /false positive percentages), *execution speed*, and *the number of memory accesses per packet*. To verify the accuracy results, we also implemented a naive algorithm to record per-flow volumes, and then find the heavy changes as the ground truth. The real positive percentage is the number of true positives reported by the detection algorithm divided by the number of real heavy change keys. The false positive percentage is the number of false positives output by the algorithm divided by the number of keys output by the algorithm. Each experiment is run 10 times with different datasets (*i.e.*, different 5-minute intervals) and the average is taken as the result.

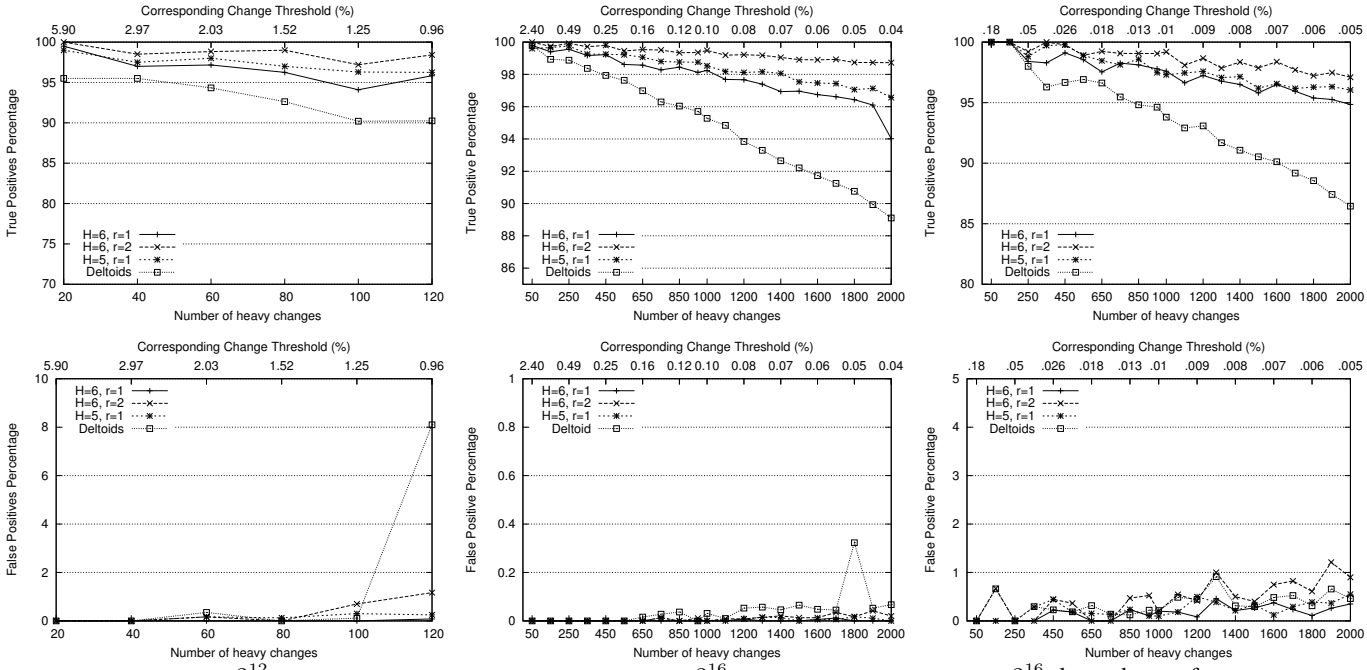
C. Software Simulation Results

1) Highly Accurate Detection Results: First, we test the performance with varying m, H and r selected before. We also vary the number of true heavy keys from 1 to 120 for $m = 4K$, and from 1 to 2000 for $m = 64K$ by adjusting ϕ. Both of these limits are much larger than the $m^{2/q}$ bound and thus are achieved using the iterative approach of Section IV-F.

As shown in Figure 7, all configurations produce very accurate results: over a 95% true positive rate and less than a 0.25% false positive rate for $m = 64K$, and over a 90% true positive rate and less than a 2% false positive rate for $m = 4K$. Among these configurations, the $H = 6$ and $r = 2$ configuration gives the best result: over a 98% true positive and less than a 0.1% false positive percentage for $m = 64K$, and over a 95% true positive and less than a 2% false positive percentage for $m = 4K$. When using the same amount of memory for recording, our scheme is much more accurate than the *deltoids* approach. Such trends remain for the stress tests and large key space size test discussed later. In each figure, the x-axis is the number of heavy change keys and their corresponding change threshold percentage ϕ.

Note that an increase of r, while being less than $\frac{H}{2}$, improves the true positive rate quite a bit. It also increase the false positive rate, but the extra original k-ary sketch bounds the false positive percentage by eliminating false positives during verification. The running time also increases for bigger r, but only marginally.

2) Iterative Approach Very Effective: As analyzed in Section IV-C, the running time grows exponentially as t exceeds



$m = 2^{12}$ $m = 2^{16}$ $m = 2^{16}$, large dataset for stress tests

Fig. 7. True positive and false positive percentage results for 12 bit buckets, 16 bit buckets, and a large dataset.

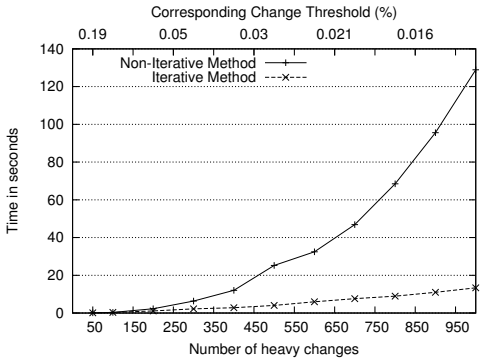


Fig. 8. Performance comparison of iterative vs. non-iterative methods

$m^{2/q}$. Otherwise, it only grows linearly. This is indeed confirmed with our experimental results as shown in Figure 8. For the experiments, we use the best configuration from previous experiments: $H = 6$, $m = 64K$, and $r = 2$. Note that the point of deviation for the running time of the two approaches is at about $250 \approx m^{2/q}(256)$, and thus matches very well with the theoretic analysis.

We implement the iterative approach by finding the threshold that produces the desired number of changes for the current iteration, detecting the offending keys using that threshold, removing those keys from the sketch, and repeating the process until the threshold equals the original threshold. Both the iterative and non-iterative approach have similarly high accuracy as in Figure 7.

3) Stress Tests with Larger Dataset Still Accurate: We further did stress tests on our scheme with two 2-hour netflow traces and detected the heavy changes between them. Each trace has about 240 GB of traffic. Again, we have very high accuracy for all configurations, especially with $m = 64K, H = 6$ and $r = 2$, which has over a 97% real positive percentage and less than a 1.2% false positive percentage as in Figure 7.

4) Performs Well on Different Networks: From Figure 6 it is evident that the data characteristics of both the ISP and the Northwestern data set are very similar, so it is no surprise that we get very close results on both data sets. Here, we omit the figures of the Northwestern data set in the interest of space.

5) Scalable to Larger Key Space Size: For 64-bit keys consisting of source IP and destination IP addresses we tested with up to the top 300 changes. Various settings give good results. The best results are for $H = 6$ and $r = 1$ with a true positive percentage of over 99.1% and a false positive percentage of less than 1.2%.

6) Few Memory Accesses Per Packet Recording: It is very important to have few memory accesses per packet for online traffic recording over high-speed links. For each packet, our traffic recording only needs to 1) look up the mangling table (see Section III-B) and 2) update each hash table in the reversible and verifier sketch. ($2H$ accesses).

Key length $\log n$ (bits)	32	64	104
# of mangling table lookup, g (# of characters in each key)	4	8	13
Size of characters in each key, c	8	8	13
Mangling table size ($2^c \times g \times 4Byte$)	4KB	8KB	13KB
memory access/pkt ($g + 2H$)	14-16	18-20	23-25
Avg memory access/pkt **(deltoids)** ($2 \times (\log n/2 + 1)$)	34	66	106

TABLE IV

MEMORY ACCESS COMPARISON: REVERSIBLE SKETCH & DELTOIDS. 104 BITS FOR 5 TUPLES (SRC IP, DEST IP, SRC PORT, DEST PORT, PROTOCOL)

For deltoids, for each entry in a hash table, there are $\log n$ counters (*e.g.*, 32 counters for IP addresses) corresponding to each bit of the key. Given a key, the deltoids data structure

needs to update each counter corresponding to a "1" bit in the binary expansion of the key, as well as update a single sum counter. Thus, on average, the number of counters to be updated is half of the key length plus one. As suggested in [3], we use 2 hash tables for deltoids. Thus, the average number of memory accesses per packet is the same as the key length in bits. The comparison between the reversible sketch and deltoids is shown in Table IV. Our approach uses only 20-30% of the memory accesses per packet as that of the deltoids, and even fewer for larger key spaces.

7) Monitoring and Detection with High Speeds: In this section, we show the running time for both recording and detection in software.

With a Pentium IV 2.4 GHz machine with normal DRAM memory, we record 2.83M items in 1.72 seconds, *i.e.*, 1.6M insertions/second. For the worst case scenario with all 40-byte packets, this translates to around 526 Mbps. These results are obtained from code that is not fully optimized and from a machine that is not dedicated to this process. Our change detection is also very efficient. As shown in Figure 8, for K=65,536, it only takes 0.34 second for 100 changes. To the extreme case of 1000 changes, it takes about 13.33 seconds.

In summary, our evaluation results show that we are able to infer the heavy change keys solely from the k-ary sketch accurately and efficiently, without explicitly storing any keys. Our scheme is much more accurate than *deltoids*, and has far fewer memory accesses per packet, even to an order of magnitude.

VII. Related Work

Most related work has been discussed earlier in this paper. Here we briefly examine a few remaining works.

Given today's traffic volume and link speeds, it is either too slow or too expensive to directly apply existing techniques on a per-flow basis [4], [1]. Therefore, most existing high-speed network monitoring systems estimate the flow-level traffic through packet sampling [23], but this has two shortcomings. First , sampling is still not scalable; there are up to 2^{64} simultaneous flows, even defined only by source and destination IP addresses. Second, long-lived traffic flows, increasingly prevalent for peer-to-peer applications [23], will be split up if the time between sampled packets exceeds the flow timeout. Thus, the application of sketches has been studied quite extensively [9], [5], [6].

The AutoFocus system automates the dynamic clustering of network flows which exhibit interesting properties such as being a heavy hitter. But this system requires large memory and can only operate offline [24]. Recently, PCF has been proposed for scalable network detection [25]. It uses a similar data structure as the original sketch, and is not reversible. Thus, even when attacks are detected, attacker or victim information is still unknown, making mitigation impossible.

VIII. Conclusion

In this paper, we propose efficient *reversible hashing* schemes which record massive network streams over high-speed links online, while maintaining the ability to detect heavy changes and infer the keys of culprit flows in (nearly) real time. This scheme has a very small memory usage and a small number of memory accesses per packet, and is further scalable to a large key space. Evaluations with real network traffic traces show that the system has high accuracy and speeds. In addition, we designed a scalable network intrusion and mitigation system based on the reversible sketches, and demonstrate that it can detect almost all SYN flooding attacks and port scans that can be found with complete flow-level logs. Moreover, we will release the software implementation soon.

References

[1] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. of ACM SIGCOMM IMC*, 2003.

[2] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *ACM SIGCOMM IMC*, 2004.

[3] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," in *Proc. of IEEE Infocom*, 2004.

[4] C. Estan et al., "New directions in traffic measurement and accounting," in *Proc. of ACM SIGCOMM*, 2002.

[5] Graham Cormode et al., "Finding hierarchical heavy hitters in data streams," in *Proc. of VLDB 2003*, 2003.

[6] G. Cormode et al., "Holistic UDAFs at streaming speeds," in *Proc. of ACM SIGMOD*, 2004.

[7] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. of IEEE VLDB*, 2002.

[8] R. S. Tsay, "Time series model specification in the presence outliers," *Journal of the American Statistical Association*, vol. 81, pp. 132141, 1986.

[9] G. Cormode and S. Muthukrishnan, "Improved data stream summaries: The count-min sketch and its applications," Tech. Rep. 2003-20, DIMACS, 2003.

[10] Philippe Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[11] A. C. Gilbert et al., "QuickSAND: Quick summary and analysis of network data," Tech. Rep. 2001-43, DIMACS, 2001.

[12] Yan Gao, Zhichun Li, and Yan Chen, "Towards a high-speed router-based anomaly/intrusion detection system," Tech. Rep. NWU-CS-05-011, Northwestern University, 2005.

[13] Muthukrishnan, "Data streams: Algorithms and applications (short)," in *Proc. of ACM SODA*, 2003.

[14] G. Cormode and S. Muthukrishnan, "Estimating dominance norms on multiple data streams," in *Proceedings of the 11th European Symposium on Algorithms (ESA)*, 2003, vol. 2461.

[15] Charles Robert Hadlock, *Field Theory and its Classical Problems*, Mathematical Association of America, 1978.

[16] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, Peter Dinda, M. Kao, and G. Memik, "Reverse hashing for high-speed network monitoring: Algorithms, evaluation, and applications," Tech. Rep. 2004-31, Northwestern University, 2004.

[17] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[18] N. weaver, S. Staniford, and V. Paxson, "Very fast containment of scanning worms," in *usenix security symposium*, 2004.

[19] H. Wang, D. Zhang, and K. G. Shin, "Detecting SYN flooding attacks," in *Proc. of IEEE INFOCOM*, 2002.

[20] H. Wang, D. Zhang, and K. G. Shin, "Change-point monitoring for detection of DoS attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 4, 2004.

[21] Xilinx Inc., "SPEEDRouter v1.1 product specification," 2001.

[22] Syplicity Inc., "Synlipfy Pro," http://www.synplicity.com.

[23] N. Duffield et al., "Properties and prediction of flow statistics from sampled packet streams," in *ACM SIGCOMM IMW*, 2002.

[24] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *Proc. of ACM SIGCOMM*, 2003.

[25] R. R. Kompella et al., "On scalable attack detection in the network," in *Proc. of ACM/USENIX IMC*, 2004.