Reverse Hashing for Sketch-based Change Detection on High-speed Networks

Robert Schweller, Yan Chen, Elliot Parsons, Ashish Gupta, Gokhan Memik and Yin Zhang

***Abstract*—**

With the ever-increasing link speeds and traffic volumes of the Internet, monitoring and analyzing network traffic usage becomes a challenging but essential service for network administrators of large ISPs or institutions. There are two popular primitives for efficient analysis over massive data streams: heavy hitter detection and heavy change detection. Although numerous approaches have been proposed for efficient heavy hitter detection [1], [2], [3], [4], [5], the sketch-based scheme [6] is one of the very few that can detect heavy changes and anomalies over massive data streams at network traffic speeds. However, sketches do not preserve keys (*e.g.*, source IP address) of the flows. Thus even if anomalies are detected, it is difficult to infer the culprit flows.

To address this challenge, we propose efficient *reversible hashing* schemes to infer the keys of culprit flows from sketches with negligible extra memory and few extra memory accesses for recording streaming data - implementing on a single FPGA board, we can achieve a throughput of over 16Gbps for all 40-byte-packet streams (the worst case traffic). Meanwhile, the heavy change detection daemon runs in the background with space complexity and computational time sublinear to the key space size. Evaluation with traces from a large edge router show that we can infer the keys for even 1,000 heavy changes while achieving over a 99% real positive percentage and less than a 0.5% false positive percentage in 22 seconds.

***Index Terms*—Network measurements, Combinatorics, and Statistics**

I. Introduction

Today's ever-increasing link speeds and traffic volumes of the Internet make monitoring and analyzing network traffic usage a challenging but essential service for managing large ISPs. Such service is important for accounting, provisioning, traffic engineering, scalable queue management and anoamly/intrusion detection [7], [2], [6]. There are two popular primitives for massive data stream analysis: heavy hitter detection and heavy change detection. The former detects any flows which constitute more than a given threshold fraction of the total traffic stream. The latter detects flows whose size changes significantly from one period to another. There are quite a few existing works for efficient and online heavy hitter detection [2], [3], [4], [5].

However, efficient online heavy change detection remains a challenging problem. Essentially, heavy change detection is more generic and more powerful than heavy hitter detection. It spans from simple absolute or relative changes, to variational changes and linear transformation of these changes for various time-series forecasting models [6], [8], [9], [10], [11]. Our goal is to *design efficient data structures and algorithms to achieve close to real-time monitoring and heavy change detection of large massive data streams, and then push for real-time operations when assisted with hardware implementation at reasonable costs*. As in [7], [2], the performance constraints for such a system are two-fold: 1) small amount of memory usage (to be implemented in SRAM) and 2) small number of memory accesses per packet.

Sketches, an emerging compact data structure, have proven to be useful in many data stream computation applications [12], [13], [5], [14]. Recent work on a variant of sketch, namely 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 [6]. It is also flexible and can be applied with various forecast models to detect anomalies.

As modelled in Section II-A, the streaming data can be viewed as a series of (*key, value*) pairs where the key can be a source IP address, or the pair of IP addresses, and the value can be the number of bytes or packets, etc. For any given key, sketch can indicate if it exhibits big change, and if so, give an accurate estimation of such 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. This means that to compare two streams, we have to know which items (keys) to query to find the streams with big changes [6], [7]. This would require either exhaustively testing all possible keys, or recording and testing all data stream keys and corresponding sketches. Unfortunately, neither of these are scalable.

Recently, Cormode and Muthukrishnan proposed *deltoids* approach for heavy change detection [7]. Though 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 address), 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, thus violating both of the aforementioned performance constraints. Moreover, this approach may not be applicable for implementation in hardware (see Section VI-A).

To address these problems, in this paper, we propose novel and efficient techniques to *reverse* sketches, focusing primarily on the k-ary sketch [6]. The observation is that only streaming data recording needs to done continuously in real-time, while the change/anomaly detection can run in the background with more memory (DRAM) and at a frequency only in the order of seconds. Then the challenge is this: how to keep extremely fast data recording while still being able to detect the heavy change keys with reasonable speed and high accuracy? Our solution has two parts as follows.

First, we include *IP mangling* and *modular hashing* in the data recording operation with negligible extra memory consumption (4KB - 8KB) and few (4 to 8) additional memory accesses per packet for IP mangling. This is in contrast to *deltoid* which uses a *factor* of 32 to 64 increase in memory and memory accesses from the original k-ary sketch. When implemented on a single FPGA board, we can sustain more than 16Gpbs even for all 40-byte-packet streams (the worst case traffic).

Next, we apply *bucket potential intersection, iterative detection*, and *bucket index matrix construction* and *matching* for heavy change key detection, which has both space and time complexity sub-linear to the key space size. We further equip the *reversible* k-ary sketch with an original k-ary sketch to statistically bound the false positive rate.

We implemented and evaluated our system with network traces obtained from a large edge router with an OC-12 link. For inferring even 1,000 heavy change keys, our schemes find more than 99% of the heavy change keys with less than a 0.5% false positive rate within 22 seconds. We further stress test our schemes with aggregated 2-hour traffic and with 64 bit key spaces. For both we achieve similar results.

The rest of the paper is organized as follows. We give an overview on the data stream model and k-ary sketch in Section II. In section Section III we discuss the algorithms for streaming data recording and in Sections IV and V discuss those for heavy change detection. We evaluate our system in Section VI, survey the related work in Section VII, and finally conclude in Section VIII.

II. Overview

A. Data Stream Model and the k-ary Sketch

Among the multiple data stream models, one of the most general is the Turnstile Model [15]. 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.

K-ary sketch is a powerful data structure to efficiently keep accurate estimates of the signals $U[a]$. A k-ary sketch consists of H hash tables of size K . The hash functions for each table are chosen independently at random from a class of hash functions from $[n]$ to $[K]$. From here on we will use the variable $m = K$ interchangeably with K. We store the data structure as a $H \times K$ table of registers $T[i][j]$ $(i \in [H], j \in [K])$. Denote the hash function for the i^{th} table by h_i. Operations on the sketch include INSERT(a, u) and ESTIMATE(a). Given a data key and an update value, INSERT(a,u) increments the count of bucket $h_i(a)$ by u for each hash table h_i. Let SUM $= \sum_{j \in [K]} T[0][j]$ be the sum of all updates to the sketch. The operation ESTIMATE(a) for a given key a returns the following.

$$v_a^{est} = \texttt{median}_{i \in [H]} \{v_a^{h_i}\} \tag{1}$$

where

$$v_a^{h_i} = \frac{T[i][h_i(a)] - \frac{SUM}{K}}{1 - 1/K}$$

If the hash functions in the sketch are 4-universal, this estimate gives an unbiased estimator of the signal $U[a]$ with variance inversely proportional to $(K-1)$ [6]. See [6] for details on the appropriate selection of H and K to obtain accurate estimates.

B. Change Detection

K-ary sketches can be used in conjunction with various forcasting models to perform sophisticated change detection as discussed in [6]. 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 inputs sets 1 and 2, if the signal for a key x is 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 $D[x] \geq \phi \cdot D$.

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 allow us to add or subtract sketches to find the sum or difference of different sketches. Any key whose estimate value in S_d that exceeds the threshold $\phi \cdot SUM = \phi \cdot D$ is denoted as a *suspect* heavy key in sketch S_d and offered as a proposed element of the set of heavy change keys.

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

More generally, we can think of our input as a sketch T in which certain buckets in each hash table are marked as *heavy*. According to formula (1) the goal is thus 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 discuss how we solve this problem in the case that $t = 1$. In section V we generalize this method to the case of larger t.

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 $U_d[x] = |U_2[x] - U_1[x]|$. Denote the total difference by $D = \sum_{x \in [n]} D[x]$. The following theorem relates the size of the sketch (in terms of K and H) with the probability of a key being incorrectly categorized as a heavy change key or not.

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

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

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 K and H. We omit the proof of this theorem in the interest of space, but refer the reader to [7] in which a similar theorem is proven.

As we discuss in IV, our reversible k-ary sketch does not have 2-universality. However, we use a second original 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 the false positives of theorem 1. 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 ϕ, we could detect the set of suspect keys for the initial sketch with respect to $\phi - \alpha$ 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 .

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	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

E. Architecture

Our change detection system has two parts as in Fig. 1: streaming data recording and heavy change detection.

Next, we will introduce each part.

III. STREAMING DATA RECORDING

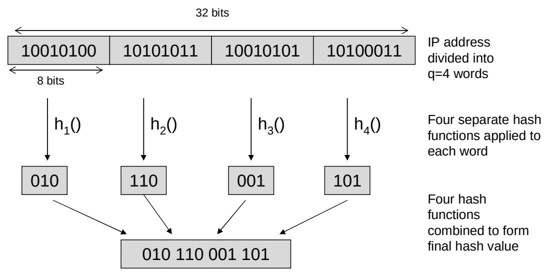

32 bits
10010100
10101011
10010101
10100011
IP address divided into q=4 words
8 bits
h1()
h2()
h3()
h4()
Four separate hash functions applied to each word
010
110
001
101
Four hash functions combined to form final hash value
010 110 001 101

Fig. 2. Modular hashing uses q hash functions to hash each word of the key , which are are then combined to form the final hash

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.

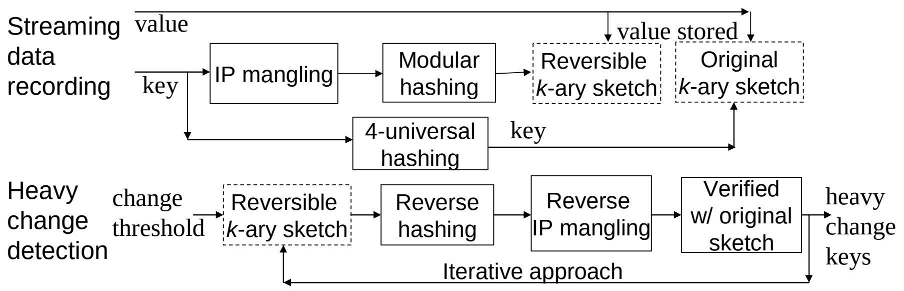


Fig. 1. Architecture of the reversible k-ary sketch based heavy change detection system for massive data streams

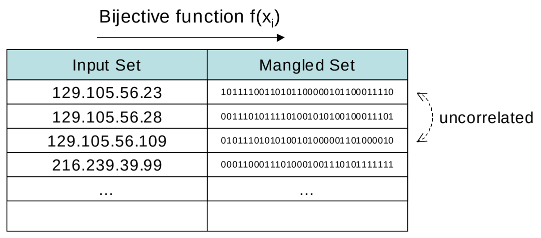
A. Modular hashing

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 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 our update time from $O(H)$ to $O(q \cdot H)$. On the other hand, as we will discuss in sections IV and V, 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 4-universal hash functions, which can map the input keys uniformly over the buckets. Though theoretically we cannot achieve the 4-universal property with modular hashing, we strive to improve modular hashing so that it works well in practice. 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. Our 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 our modular hashing scheme with the network traces used for evaluation (see Section VI), the distribution of the number of keys per bucket was highly skewed, with most of the IP addresses going to a few buckets (Figure 4). This significantly disrupts the estimation accuracy of our reversible k-ary sketch. To overcome this problem, we introduce the technique of *IP mangling*.

B. IP Mangling

In IP mangling we attempt to artificially randomize the input data in an attempt to destroy any correlation or spa-


Bijective function f(x_i)
uncorrelated

Input Set	Mangled Set
129.105.56.23	10111100110101100000101100011110
129.105.56.28	00111010111101001010100100011101
129.105.56.109	01011101010100101000001101000010
216.239.39.99	00011000111010001001110101111111
...	...

Fig. 3. IP-mangling destroys any correlation between the input data to present completely random keys to the modular hash functions.

tial 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 $[n]$ (Figure 3). 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.

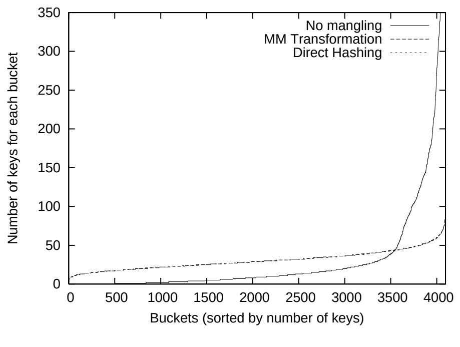


Fig. 4. Distribution of number of keys for each bucket under three hashing methods.

Our choice of the bijective function f is based on simple arithmetic operations on a Galois Extension Field [16] $\mathbb{GF}(2^\ell)$, 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. We believe that such a mapping will sufficiently alter the original set of keys such that the locality (in terms of hamming distance or absolute difference) of streaming keys will be destroyed. 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: $x = \overline{x_3x_2x_1x_0}$. According to the finite field arithmetic, we have $a \otimes x = a \otimes \overline{x_3x_2x_1x_0} = \bigoplus_{i=0}^{3} a \otimes (x_i \ll 8\, i)$, 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 the fast memory ($2^8 \times 4 \times 4Bytes = 4KB$ for 32-bit IP address). Note that only IP mangling needs extra memory and extra memory lookup, the 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.

We find that in practice, a bijective function with this property 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 MM transformation for IP mangling, and 3) direct hashing (a completely random hash function). Figure 4 shows the distribution of the number of keys per bucket for each hashing scheme. We observe that the key distribution of modular hashing with MM transformation is almost 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.

IV. Reverse Hashing: $t = 1$

Now we discuss how to perform the detection phase of the change detection process. As discussed in the overview, our approach is to take a k-ary sketch that has been updated with all data items in the input data stream and output all suspect keys with respect to the sketch. That is, we solve the reverse sketch problem with respect to the given sketch. The reverse sketch problem can always be solved in $O(n \cdot H)$ run time by simply testing all possible keys in the space $[n]$. However, the key space n is typically prohibitively large. We thus wish to solve the problem in time sub-linear in n. We show how this can be done when the hash functions for the sketch are modular. In this section we discuss how to solve the problem with input t equal to 1 using our basic approach of taking modular bucket intersections. We then generalize our method to larger t in section V.

A. Modular Bucket Intersections

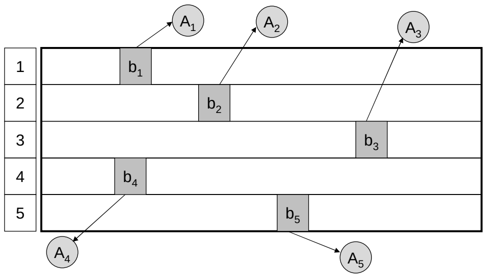


Fig. 5. Each heavy change bucket reverse maps to a set of IP addresses, *e.g.*, $\simeq 2^{20}$ for our example k-ary sketch

At $t = 1$ we have that there is at most one heavy bucket in each hash table. For a given heavy bucket in hash table i, suppose we can obtain the set A_i consisting of all keys that hash to the heavy bucket in the i^{th} hash table (Figure 5). We call this set the bucket's *bucket potentials*. For $r = 0$ we can determine the suspect keys by taking the intersection $\bigcap_{i=0}^{H-1} A_i$ of the bucket potentials. For $r > 0$ we can do a modified intersection that is still quite simple. However, each set A_i is expected to contain $\frac{n}{m}$ elements. The key space n is assumed to be prohibitively large. It is thus difficult to efficiently obtain the sets A_i, and more difficult to take the H-wise intersection of such large sets efficiently. For the example of 32 bit IP address keys with $m = 2^{12}$ we are dealing with $\frac{n}{m} = 2^{20}$. To determine which 2^{20} elements are in each set A_i, and to intersect H sets of size 2^{20} is too taxing on memory and speed.

Modular hashing solves this efficiency problem. First, to determine the sets A_i we can store a many-to-one reverse lookup table for each hash function h_i. Without modular hashing this would require $\Theta(n \cdot \log m)$ storage space for each hash function. But with modular hashing we can implicitly store a reverse lookup table for h_i by storing the smaller reverse lookup tables of each of its q modular hash functions $h_{i,w}$. That is, we store a many-to-one reverse lookup table $h_{i,w}^{-1}[j]$ that maps each key $j \in [m^{\frac{1}{q}}]$ to a list of $(\frac{n}{m})^{\frac{1}{q}}$ distinct values from the set $[n^{\frac{1}{q}}]$. We can store such reverse lookup tables in $O(\frac{1}{q}(\log m)n^{\frac{1}{q}})$ space. This gives a total space complexity of $O(H(\log m)n^{\frac{1}{q}})$ for the $H \cdot q$ hash functions. Depending on the choice of q, this offers various levels of improvement in space usage over the original $\Theta(n \cdot \log m)$.

Modular hashing also allows for more efficient intersection of sets of bucket potentials A_i. For each given bucket the reverse lookup table gives us q sets of size

$(\frac{n}{m})^{\frac{1}{q}}$ corresponding to what we call the *modular* bucket potentials of each word. Denote the modular bucket potential set for hash table i and word w as $A_{i,w}$. These sets give a compact representation of the set of bucket potentials because a key x is in A_i if and only if the w^{th} word of x is in $A_{i,w}$ for each w from 1 to q. In addition, these modular potential sets are only of size $(\frac{n}{m})^{\frac{1}{q}}$, compared to the size $\frac{n}{m}$ potential sets. For $n = 2^{32}$, $m = 2^{12}$, and $q = 4$, the modular potential sets are of size only $\frac{2^8}{2^3} = 32$ each. This reduces the time for taking the H intersections of the bucket potential sets A_i by allowing q separate H-wise intersections of the smaller sets $A_{i,w}$. For example, suppose a heavy bucket has the modular potentials sets $A_{(i,1)}, A_{(i,2)}, A_{(i,3)}, A_{(i,4)}$ for $q = 4$. In the case of $r = 0$ and $H = 5$ the intersection involves four separate intersection operations: $X_j = A_{(1,j)} \bigcap A_{(2,j)} \bigcap A_{(3,j)} \bigcap A_{(4,j)} \bigcap A_{(5,j)}$ for $j = 1, 2, 3, 4$, corresponding to four partitions of the IP address. The resultant intersections from the four partitions can then be combined to form the final set of suspect keys, i.e, any $x_1.x_2.x_3.x_4$ such that each $x_j \in X_j$. Since each set being intersected has size $(\frac{n}{m})^{\frac{1}{q}}$ we can determine these q different sets of H set intersections in time $O(q \cdot H(\frac{n}{m})^{\frac{1}{q}})$. In section V-D we discuss how to choose the value for q with respect to n to make our algorithms run as efficiently as possible.

V. Reverse Hashing: General Case

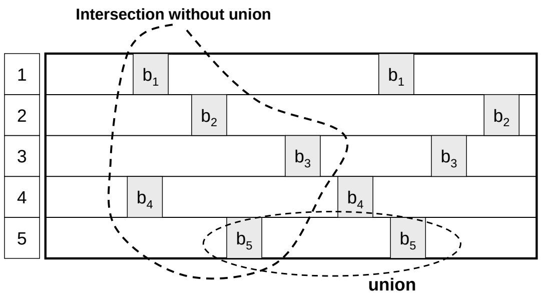


Fig. 6. For the case of $t = 2$, various possibilities exist for taking the intersection of each bucket's potential keys

We now generalize our method of reverse hashing to the case where there are multiple heavy buckets in each hash table.

We use techniques similar to the modular bucket intersections for $t = 1$. However, for $t \geq 2$, the technique must be extended. To understand the problem, consider the simple case of $t = 2$, as shown in Figure 6. 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. The case for $t \geq 2$ is thus much more difficult than for $t = 1$. 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

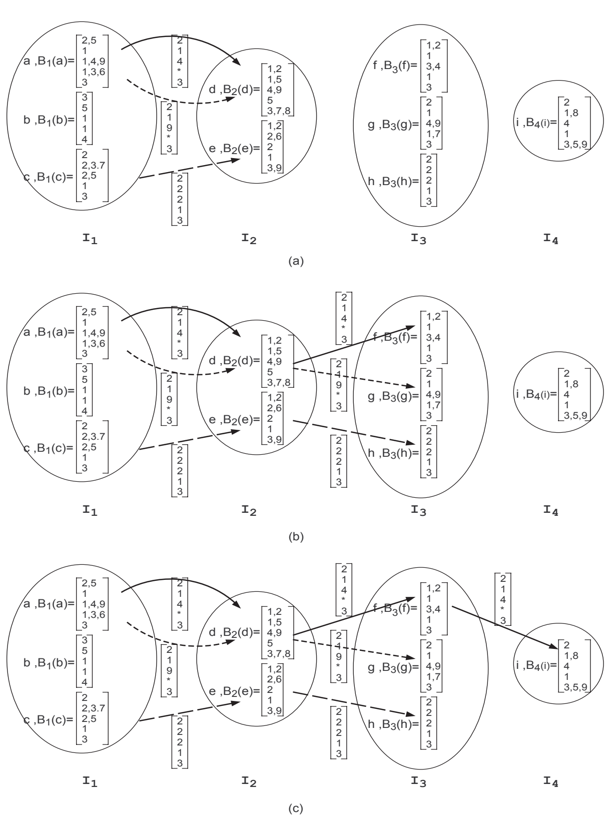


Fig. 7. Given the q sets I_w and bucket index matrices B_w we can compute the sets A_w incrementally. The set A_2 containing $(\langle a,d\rangle, \langle 2,1,4,*,3\rangle)$, $(\langle a,d\rangle, \langle 2,1,9,*,3\rangle)$, and $(\langle c,e\rangle, \langle 2,2,2,1,3\rangle)$ is depicted in (a). From 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).

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 $P = NP$.

However, if we choose random modular hash functions as described in IV we can solve the problem efficiently with high probability as discussed later. Next, we present an algorithm to solve the reverse sketch problem for any t that is assured to obtain the correct solution with a polynomial run time in q and H with very high probability.

A. Notation

To describe the algorithm we use, 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 $h^{-1}_{i,w}[j] = \{x \in [n^{\frac{1}{q}}] \mid h_{i,w}(x) = j\}$.

Let $I_w = \{x \mid x \in \bigcup_{j=0}^{t_i-1} h^{-1}_{i,w}[\sigma_w(t_{i,j})]$ 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 this set as the *intersected modular potentials* for word w. For instance, in Figure 7, 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 1st, 3rd, and 8th 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 7.

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 7, $B_1(a) \bigcap^r B_2(d) = \langle\{2\},\{1\},\{4\},\{*\},\{3\}\rangle$, 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 7 for example, $A_4 = \langle a, d, f, i\rangle, \langle 2, 1, 4, *, 3\rangle$ is the suspect key. Each element of A_w can be denoted as a *path* in Figure 7. 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.

B. 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 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 7, 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 $h^{-1}_{i,w}[\sigma_w(t_{i,j})][k]$ 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 $A_{w+1}$.
```

C. 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.

Proof: For simplicity, below we assume $r = 0$. (The proof for $r > 0$ is similar but slightly more involved.)

For any vector $a \in [n^{\frac{1}{q}}]^2$, $b \in [m^{\frac{1}{q}}]^2$, $u \in [t]^H$, define

$$Y_{a,b}^u = \begin{cases} 1 & \delta_w(t_{i,u[i]}) = h_{i,w}(a[w]) = b[w] \\ & \qquad \text{for } \forall i \in [H], \forall w \in [2], \\ 0 & \text{otherwise} \end{cases}$$

Clearly, A_2 has $Y = \sum_{a,b,u} Y_{a,b}^u$ bucket index vectors.

We have Prob$\{h_{i,w}(a[w]) = b[w]\} = m^{-1/q}$. With mangling, we have Prob$\{\delta_w(t_{i,u[i]}) = b[w]\} = m^{-1/q}$. Therefore, $E(Y_{a,b}^u) = (m^{-2/q})^{H+1}$. This implies

$$E(Y) = \sum_{a,b,u} E(Y_{a,b}^u) = n^{2/q} \cdot (t \cdot m^{-2/q})^H$$

We now estimate $Var(Y)$. For any $a, c \in [n^{\frac{1}{q}}]^2$, $u, v \in [t]^H$, define $e(a,c) \equiv |\{w | w \in [2] \wedge a[w] = c[w]\}|$, and $e(u,v) \equiv |\{i | i \in [H] \wedge u[i] = v[i]\}|$.

We have

$$E(Y_{a,b}^u \cdot Y_{c,d}^v) = \begin{cases} 0 & \\ & b \neq d \wedge (e(u,v) \neq 0 \vee e(a,c) \neq 0) \\ (m^{-2/q})^{2H+2} & \\ & b \neq d \wedge e(u,v) = 0 \wedge e(a,c) = 0 \\ (m^{-2/q})^{2H+2-j-k} & \\ & b = d \wedge e(u,v) = j \wedge e(a,c) = k \end{cases}$$

Therefore,

$$\begin{aligned} Var(Y) &= E(Y^2) - E(Y)^2 \\ &= \sum_{a,b,u,c,d,v} E(Y_{a,b}^u \cdot Y_{c,d}^v) - E(Y)^2 \\ &= -E(Y)^2 + \sum_{\substack{a,b,c,d,u,v \\ b \neq d \wedge e(u,v)=0 \wedge e(a,c)=0}} (m^{-2/q})^{2H+2} + \\ & \qquad \sum_{\substack{a,b,c,d,u,v,j,k \\ b \neq d \wedge e(u,v)=j \wedge e(a,c)=k}} (m^{-2/q})^{2H+2-j-k} \\ &\equiv -E(Y)^2 + T_1 + T_2 \end{aligned}$$

We can prove

$$T_1 \leq \sum_{a,b,c,d,u,v} (m^{-2/q})^{2H+2} = E(Y)^2$$

$$T_2 \leq n^{4/q} \cdot (\frac{t}{m^{2/q}})^H \cdot (1 + \frac{t}{m^{2/q}})^H$$

With $\frac{t}{m^{2/q}} \leq \frac{\sqrt{5}-1}{2}$, we have $E(Y) < n^{2/q}$ and $Var(Y) \leq T_2 \leq n^{4/q}$. By Chebyshev Inequality, we can then show that the number of bucket index vectors in A_2 is $O(n^{2/q})$ with high probability. ■

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 V-E, 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, the running times for MODULAR_POTENTIALS and step 2 of REVERSE_HASH is $O(n^{2/q})$. The running time of EXTEND is $O(n^{3/q})$. So the total running time is $O((q-2) \times n^{3/q})$.

D. 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.

However, we must also assure that our k-ary sketch maintains a large degree of accuracy. In particular, there are two constraints we must adhere to. First, we need it to be very unlikely that two given keys hash to the same bucket in all H hash tables. Thus, for a given choice of one bucket from each hash table, we want the expected number of keys that hash to the buckets, $\frac{n}{m^H}$ for completely random hashing, to be sufficiently small. Second, we cannot allow the the size of the space that the modular keys map to, $m^{\frac{1}{q}}$, to drop below 2. To handle larger values for t, we need to keep $m^{\frac{1}{q}}$ even larger than 2 (Subsection V-C). For given constants ϵ and c we summarize our two constraints as follows.

1) $\frac{n}{m^H} < \epsilon$, equivalent to $(\frac{n}{\epsilon})^{\frac{1}{H}} < m$.

2) $m^{\frac{1}{q}} > c$, equivalent to $m > c^q$

First note that for $m \geq \log n$, constraint one is fulfilled for $H = \Theta(\frac{\log n}{\log\log n})$. Both of these values are sufficiently small in n, so constraint (1) is easy to fulfill. We thus focus on constraint (2), which boils down to choosing a value for q and making the other parameters as small as that choice allows. We consider two strategies.

Solution 1: Our first solution attempts to minimize the size of the largest of the four terms. This is accomplished by setting $q = \sqrt{\log n}$. This yields:

$$q = \sqrt{\log n} \qquad m = c^{\sqrt{\log n}}$$
$$n^{\frac{1}{q}} = 2^{\sqrt{\log n}} \quad H = O(\sqrt{\log n})$$

For $c = 2$, m and $n^{\frac{1}{q}}$ are of the same complexity and dominate the four values we are interested in. Any alternate choice of q would raise the value of either m or $n^{\frac{1}{q}}$. This choice of q thus minimizes the largest of the four values. This gives the optimal solution when streaming update and change detection occur at similar frequencies.

Solution 2: One issue with solution one is that the size of the hash table is more than poly-logarithmic in n. Since only the update procedure for the sketch executes at network traffic speeds, we need to have a smaller m to have the entire sketch fit into fast memory while keeping $n^{\frac{1}{q}}$ reasonably small. Our solution is to set $q = \log\log n$. This yields:

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

Having decreased the value of q from solution 1, we have made the size of the hash table $m \cdot H$ only poly-logarithmic in n. This should allow the update phase of the problem to be scalable to large n. The drawback is that the value of $n^{\frac{1}{q}}$ is increased. However, this term does not come into play until the second phase of change detection when we perform the actual detection. Since this is not performed for every packet, we can withstand larger terms for this phase. For $n = 2^{32}, 2^{64}$, the two arguably most important cases, $n^{\frac{1}{\log\log n}}$ is $2^{32/5} = 85$ and $2^{64/6} = 1626$, respectively. This is clearly quite manageable.

E. *Iterative Detection*

From our discussion in section V-C we have that our detection algorithm can only effectively handle t of size at most $m^{\frac{2}{q}}$. With our discussion in section V-D this is only a constant. To handle larger t, we propose the following heuristic. Suppose we can comfortably handle at most t' heavy buckets per hash table. If a given ϕ percentage results in $t > t'$ 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 t' 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 t' buckets from each hash table and repeat. Continue until there are no heavy buckets left that haven't been considered.

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.

VI. Implementation and Evaluation

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

A. *Hardware Implementation for Traffic Recording*

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 [17], each with 2.5M system gates contained within 9600 Configurable Logic Blocks (CLBs) interconnected via a cross-bar along with memory modules. This development board is hosted by a Solaris Ultra-10 workstation. The unit is implemented using the Synplify Pro 7.2. tool [18]. Such FPGA board only costs about $1000.

The k-ary sketch hardware consists of H hash units each of which addresses a single K-element array. For almost all configurations, delay is the bottleneck. Therefore, we have optimized it using excessive pipelining. The resulting maximum throughput for 40-byte-packet streams are presented in Table II. For the original k-ary sketch, we achieve a high bandwidth of over 22 Gbps. Even for reversible hashing with IP mangling and modular hashing, we achieve 16.2 Gbps. Note that currently, although the largest Xilinx FPGA contains a total of 10Mbits of block SRAM, due to its architecture only up to 600KBytes of this space can be efficiently utilized. Since the deltoid approach requires more than 1MB to detect 100 or more changes, it cannot even fit into the latest FPGAs.

B. *Software Simulation Methodology*

In this section we evaluate our schemes with *netflow* traffic traces collected from a large edge router. The traces are divided into five-minute intervals with the traffic size

TABLE II
MAXIMUM SUPPORTED BANDWIDTH (GBPS) FOR ALL 40-BYTE-PACKET STREAMS ($K = 4096$)

	$H = 1$	$H = 5$
Original k-ary sketch	28.800	22.656
With modular hashing	23.360	19.264
Modular hashing+ IP mangling	17.088	16.160

for each interval averaging about 7.5GB. Our metrics include *speed*, *real positive*, and *false positive percentage*. To verify our 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 refers to 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.

We use a k-ary sketch with a variety of configurations with different H, K and r. We also run our simulations both with and without the iterative approach as described in section V-E. Finally, we stress test our schemes separately with 1) two two-hour traffic files with 240 GB average volume, and 2) with 64-bit keys.

The total memory consumption for update recording is only $2 \times \langle numberoftables \rangle \times \langle numberofbins \rangle \times 4bytes/bucket$. It includes a *reversible* k-ary sketch and a *original* k-ary sketch. In our largest configuration, with 6 tables and 64K bins, our memory usage is 3MB. The smallest with 5 tables and 4K buckets only takes 160KB memory. When using 32-bit keys, we use the source IP address of each flow as the key, and set $q = 4$. For our 64 bit-key trials, we concatenate the source and destination IP addresses of a flow, and set $q = 8$.

C. Software Simulation Results

1) Accuracy performance analysis: First, we test the performance with varying m, H and r. We consider all possible combinations from: $m = 4K$ or $64K$, $H = 5$ or 6, and $r = 1$ or 2. We vary the number of true heavy keys from 1 to 140 for $m = 4K$, and from 1 to 1000 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 V-E.

As shown in Figure 8, all configurations produce very accurate results: over a 95% true positive rate and less than a 1.1% false postive rate for $m = 64K$, and over a 90% true positive rate and less than a 4% false positive rate for $m = 4K$. Among these configurations, the $H = 6$ and $r = 2$ configuration gives the best result: over a 99% true positive percentage and less than a 0.5% false positive percentage for $m = 64K$, and over a 95% true positive percentage and less than a 2% false positive percentage for $m = 4K$. 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 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 positive during verification. The running time also increases for bigger r, but only marginally.

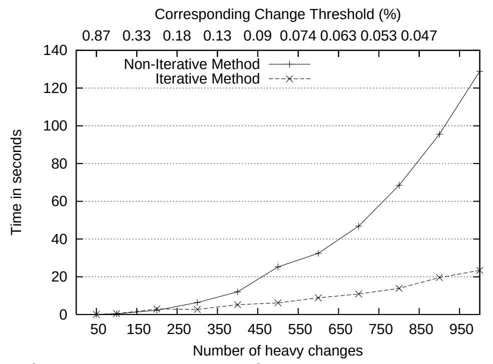


Fig. 9. Performance comparison of iterative vs. non-iterative methods

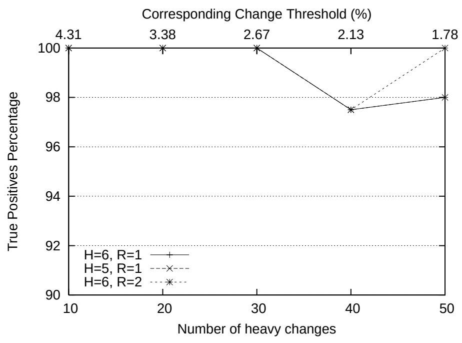


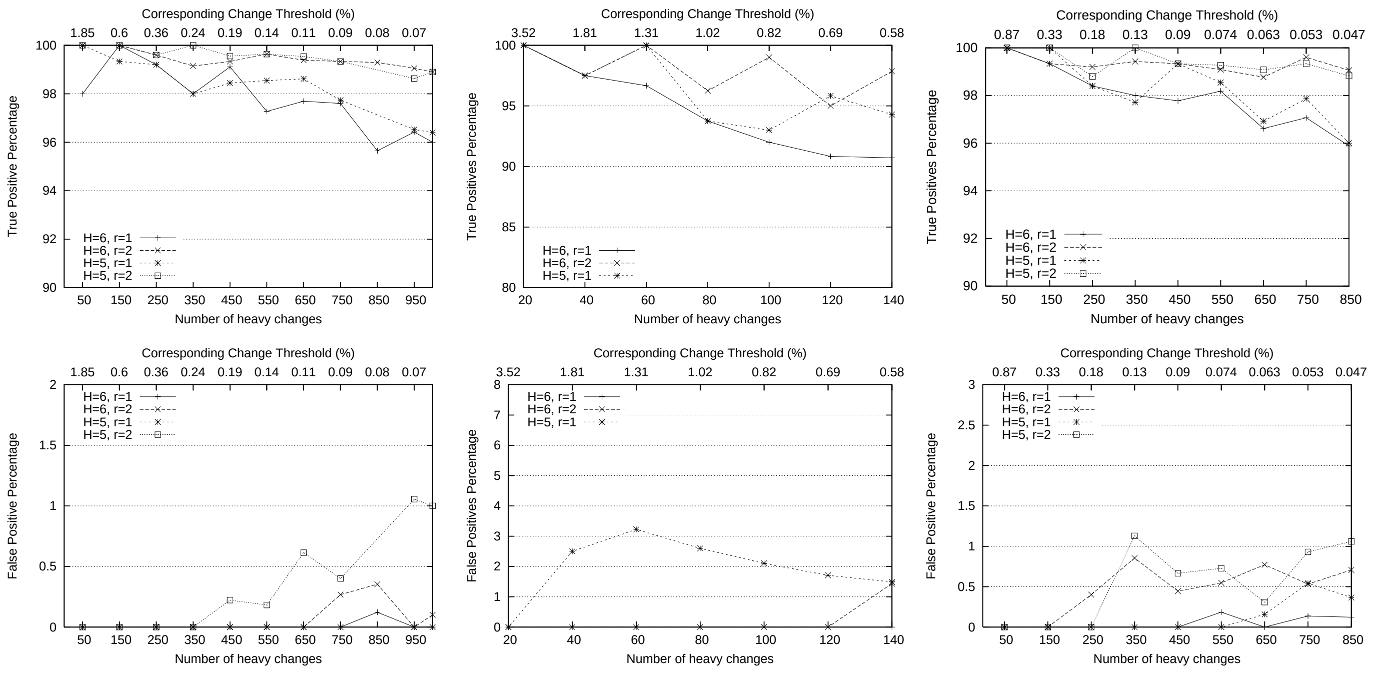
Fig. 10. Results for 64 bit keys: SrcIP and DestIP address, resulting in multi-dimensional analysis

2) Effectiveness of iterative approach: As analyzed in Section V-B, the running complexity will go exponentially high when $t > m^{2/q}$. Otherwise, it only grows linearly with t. This is indeed confirmed with our experiment results as shown in Figure 9. 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 8.

3) Stress tests with larger dataset: 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 99% real positive percentage and less than a 1% false positive percentage as in Figure 8.

4) Results on larger key space size: Figure 10 shows the effectiveness of our algorithms for 64-bit keys consisting of source IP and destination IP addresses. The true positive percentage is over 97%. The false positive rate is zero for all the configurations.



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

Fig. 8. True positive percentage and false positive percentage results for 12 bit buckets, 16 bit buckets, and a large dataset for stress tests.

5) Speed results: In section VI-A, we show that our reversible k-ary sketch in hardware can sustain 16.2Gbps throughput for recording all-40-byte packet streams. 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 2,827,318 items in 1.72 seconds, *i.e.*, 1,643,789 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 9, for $K = 65536$, it only takes 0.078 second for 50 changes, 0.42 second for 100 changes, and 2.92 seconds for 200 changes which already covers about 0.2% of the total changes. To the extreme case of 1000 changes, it takes about 22 seconds.

In short, our evaluation results show that we were able to infer the heavy change keys solely from the k-ary sketch accurately and efficiently, without explicitly storing any keys or taking a second pass over the data.

VII. Related Work

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 [2], [6]. Therefore, most existing high-speed network monitoring systems estimate the flow-level traffic through packet sampling [19], [20], 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 [19], will be split up if the time between sampled packets exceeds the flow timeout.

Applications of sketches in the data streaming community have been researched quite extensively in the past. Usually the work has focused on extracting certain data aggregation functions with the use of sketches, like quantiles and frequent items [12], distinct items [13] etc. In the context of networking, sketches have been applied to detect IP stream metrics like heavy hitters [4] and quantiles [5], [14] at networking streaming speeds.

As mentioned before, the closest work to ours is the *deltoids* approach [7]. Next, we will fully compare it with our reversible k-ary sketch.

A. *Comparison with the* deltoids *approach*

Table III lists the efficiency for both approaches for both of the two phases of change detection, the update phase and the detection phase. The complexities for the reverse sketch approach are derived using strategy 2 describe in section (V-D).

The advantage of our approach is in the update phase of the algorithm. The speed of updating the data structure per item in the stream needs to be as fast as the incoming network traffic to be applied online. The actual number of operations performed by our update versus the the deltoids approach is asymptotically the same. However, in a high speed online setting, the number of arithmetic operations performed is rarely the bottleneck for performance. A more accurate measure is the number of memory accesses and the size of the memory used. In both of these categories the reverse sketch has an advantage over the deltoids approach.

In the case of memory accesses our approach can be implemented to make only a single memory access per hash table. These accesses correspond to the insertion of the $\log m$ bit hashed key for each of the H hash tables. This is all that we need because our modular hash functions can be represented with compact, randomly seeded equations. The hashing of the $\log \log n$ modular keys per hash table thus only requires arithmetic operations, but not

TABLE III

A COMPARISON BETWEEN THE REVERSE SKETCH METHOD AND THE DELTOIDS APPROACH. HERE t DENOTES THE NUMBER OF HEAVY CHANGE KEYS IN THE INPUT STREAM.

	Update			Detection	
	memory	memory accesses	operations	memory	operations
Reverse Sketch	$\Theta(\frac{(\log n)^{\Theta(1)}}{\log\log n})$	$\Theta(\frac{\log}{\log\log n})$	$\Theta(\log n)$	$\Theta(n^{\frac{1}{\log\log n}} \cdot \log\log n)$	$O(n^{\frac{3}{\log\log n}} \cdot \log\log n \cdot t)$
Deltoids	$\Theta(\log n \cdot t)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n \cdot t)$	$O(\log n \cdot t)$

memory access. On the other hand, the deltoids approach must actually update $\Theta(\log n)$ counters in its data structure per update. It thus cannot achieve our smaller number of memory accesses.

In the case of memory usage, our approach uses memory that is constant in the number of heavy change keys t. The reason for this is because we can perform the iterative detection described in section V-E. That is, using $\Theta(\frac{(\log n)^{\Theta(1)}}{\log\log n})$ memory we can only efficiently detect a constant t' number of heavy changes. However, we can repeatedly find the approximately top t' heavy changes keys until we have efficiently obtained our finished list. Thus, while our detection run time grows linearly in t, our memory usage does not.

The deltoids approach, on the other hand, cannot use iterative detection and thus must increase the size of its data structure to detect larger numbers of heavy changes. In the case were the size of m is set to $\log n$, we get that for $t = \omega(\frac{\log}{\log\log n})$ our scheme is asymptotically superior to the deltoids method as far as memory used. We feel that such values for t are reasonable.

As a practical comparison of speed and memory usage, we consider the software implementation of our algorithm with $H = 6$ and $m = 2^{12}$. For these settings, we use about 200 KB memory, and can insert 1,643,789 items per second. We also achieved more than a 95% true positive percentage for up to 140 heavy changes. The deltoids approach only achieves an insertion rate of about 1,200,000 items per second and uses between 1 and 3 MB to detect between 100 and 200 heavy changes with accuracy above 95%. Note that a system with such size of memory cannot be implemented in a single FPGA board.

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.

VIII. CONCLUSION

Online heavy change detection is a powerful building block for network anomaly detection, but has received little attention in research except the recent k-ary sketch-based scheme proposed in [6]. However, this scheme is not reversible. Thus we propose efficient *reversible hashing* schemes to infer the keys of culprit flows from sketches with negligible extra memory and small extra memory access for recording streaming data - we obtain 16Gpbs throughput on a single FPGA board even for all 40-byte-packet streams. Evaluations with real network traffic traces show that we can infer the keys for even 1000 heavy changes with high accuracy in less than 22 seconds.

REFERENCES

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

[2] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. of ACM SIGCOMM*, 2002.

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

[4] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava, "Finding hierarchical heavy hitters in data streams," in *VLDB 2003*, 2003.

[5] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, , and D. Srivastava, "Holistic UDAFs at streaming speeds," in *Proceedings of ACM SIGMOD*, 2004.

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

[7] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," in *Proc. of IEEE Infocom*, 2004, Wed Feb 4 13:07:55 EST 2004.

[8] R. S. Tsay, "Outliers, level shifts, and variance changes time series," *Journal of Forecasting*, vol. 7, pp. 1–20, 1988.

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

[10] C. Chen and L.-M. Liu, "Joint estimation of model parameters and outlier effects in time series," *Journal of the American Statistical Association*, vol. 88, pp. 284297, 1993.

[11] C. Chen and L.M. Liu, "Forecasting time series with outliers," *Journal of Forecasting*, vol. 12, pp. 1335, 1993.

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

[13] 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.

[14] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin. J. Strauss, "QuickSAND: Quick summary and analysis of network data," Tech. Rep. 2001-43, DIMACS, 2001.

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

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

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

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

[19] Nick Duffield, Carsten Lund, and Mikkel Thorup, "Properties and prediction of flow statistics from sampled packet streams," in *Proc. of ACM SIGCOMM IMW*, 2002.

[20] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *ACM SIGCOMM*, 2003.