Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams

Robert Schweller, Ashish Gupta, Elliot Parsons, Yan Chen
Department of Computer Science
Northwestern University
Evanston, IL 60201-3150, USA
{schwellerr, ashish, e-parsons, ychen}@cs.northwestern.edu

ABSTRACT

Traffic anomalies such as failures and attacks are increasing in frequency and severity, and thus identifying them rapidly and accurately is critical for large network operators. The detection typically treats the traffic as a collection of flows and looks for heavy changes in traffic patterns (*e.g.*, volume, number of connections). However, as link speeds and the number of flows increase, keeping per-flow state is not scalable. The recently proposed sketch-based schemes [14] are among the very few that can detect heavy changes and anomalies over massive data streams at network traffic speeds. However, sketches do not preserve the key (*e.g.*, source IP address) of the flows. Hence, even if anomalies are detected, it is difficult to infer the culprit flows, making it a big practical hurdle for online deployment. Meanwhile, the number of keys is too large to record.

To address this challenge, we propose efficient *reversible hashing* algorithms to infer the keys of culprit flows from sketches without storing any explicit key information. No extra memory or memory accesses are needed for recording the streaming data. Meanwhile, the heavy change detection daemon runs in the background with space complexity and computational time sublinear to the key space size. This short paper describes the conceptual framework of the reversible sketches, as well as some initial approaches for implementation. See [23] for the optimized algorithms in details. Evaluated with netflow traffic traces of a large edge router, we demonstrate that the reverse hashing can quickly infer the keys of culprit flows even for many changes with high accuracy.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network monitoring

General Terms

Measurement, Algorithms

Keywords

Change detection, Network anomaly detection, Data stream computation, Sketch, Modular hashing, IP mangling, Reverse hashing

IMC'04, October 25–27, 2004, Taormina, Sicily, Italy.

1. INTRODUCTION

Real-time network flow monitoring at high packet rates is a challenging but crucial service for network administrators of large ISPs or institutions. Such service is important for accounting, provisioning, traffic engineering, scalable queue management, and anomaly and intrusion detection [5, 8, 14]. Take the intrusion detection systems (IDSs) for instance, most existing IDSs reside on single host or low-end routers, examining the application-level [22, 15] or system-level [13, 25] logs, or the sniffed network packets [20, 21]. However, today's fast propagation of viruses/worms (*e.g.*, Sapphire worm) can infect most of the vulnerable machines in the Internet within ten minutes [18] or even less than 30 seconds with some highly virulent techniques [24]. Thus it is crucial to identify such outbreaks in their early phases, which can only be possibly achieved by detection at routers instead of at end hosts [26].

Given today's traffic volume and link speeds, the detection method has to be able to handle potentially several millions or more of concurrent network time series. Thus it is either too slow or too expensive to directly apply existing techniques on a per-flow basis [8, 14]. The essential requirements for online flow-level monitoring on high-speed networks are two-fold: 1) small amount of memory usage (to be implemented in SRAM) and 2) small amount of memory accesses per packet [5, 8]. In response to this demand, the field of *data streaming computation* is emerging, which deals with various computations that can be performed in a space- and time-efficient fashion. Most of the existing work comes from the database and theory communities, as reviewed in a comprehensive survey [17]. Here we call for bringing techniques from these domains to bear on networking. One particularly powerful technique is the sketch [10], a probabilistic summary technique for analyzing large network traffic streams without keeping per-flow states.

Most existing research on data streaming computation focus on scalable heavy-hitter detection [8, 16, 2]. However, heavy-hitters do not necessarily correspond to flows experiencing anomalies (*e.g.*, significant changes), and thus it is not clear how their techniques can be adapted for anomaly detection. Here, we focus on a more generic and powerful primitive: heavy change detection, which spans from simple absolute or relative changes, to variational changes and linear transformation of these changes for various time-series forecasting models [14]. Recently, a variant of the sketch data structure, the k-ary sketch, was proposed as one of the first schemes for real-time heavy change detection over massive data streams [14]. As shown in Section 2, the k-ary sketch uses a constant, small amount of memory, and has constant per-record update and reconstruction cost [14].

However, one major obstacle for building anomaly/intrusion detection system on k-ary sketch is its *irreversibility*. As modeled in

Section 2.1, 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. While for any given key, sketch can indicate if it exhibits big change, and if so give an accurate estimation of such change, such process is *irreversible*. 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 [14, 5]. 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.

In this paper we focus on this problem and provide efficient algorithms to reverse sketches, focusing primarily on k-ary sketches [14]. 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: how to keep extremely fast data recording while still being able to detect the heavy change keys with reasonable speed and high accuracy? In this extended abstract, we set up the general framework for the reversible k-ary sketch, and discuss some initial approaches for implementation. The fully optimized algorithms and evaluations are presented in [23], especially for multiple heavy change detection. With no or negligible extra memory and extra memory accesses for recording streaming data, the heavy change detection daemon runs in the background with space complexity and computational time sublinear to the key space size.

The rest of the paper is organized as follows. In Section 2, we introduce the data stream model, formulate the heavy change detection problem, and present the architecture of reversible k-ary sketch system. The system has two parts: streaming data recording (Section 3) and heavy change detection (Section 4). We show some preliminary evaluation results based on a large edge router traffic data in Section 5. For detecting the keys corresponding to the top 100 changes, we achieve over 95% of true positive rate and less than 2% of false positive rate in 0.42 seconds. The streaming data are recorded with less than 200KB memory. Related work are surveyed in Section 6, and finally the paper concludes in Section 7.

2. OVERVIEW

2.1 Data Stream Model and the k-ary Sketch

Among the multiple data stream models, one of the most general is the Turnstile Model [19]. 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.

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

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 $(K-1)$ [14]. See [14] for details on the appropriate selection of H and K to obtain accurate estimates.

2.2 Change Detection Problem Formulation

K-ary sketches can be used in conjunction with various forcasting models to perform sophisticated change detection as discussed in [14]. 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 allows 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.

Instead of focusing directly on finding the set of keys that have heavy change, we instead can attempt to find the set of keys denoted as suspects by a sketch. [14, 23] discuss how to choose appropriate values for K and H so that the set of suspects is a sufficiently good approximation to the set of actual heavy change keys. For simplicity we focus on the simpler problem of finding the set of keys that hash to heavy buckets in all H hash tables. That is, we can think of our input as a sketch T in which certain buckets in each hash table are marked as *heavy*. Let t be the maximum number of distinct heavy buckets in any given hash table, we get the following **Reverse Sketch Problem**:

Input: An integer $t > 0$, a sketch T with hash functions $\{h_i\}_{i=0}^{H-1}$ from $[n]$ to $[m]$, and 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 each $i \in [H]$.

Solving the reverse sketch problem is a good way to approximate the set of heavy change keys. Consider the case in which there is exactly one heavy bucket in each hash table. The expected number of false positives (number of keys that hash to all heavy buckets by chance) for H hash tables is $E[x] = n(\frac{1}{m})^H$ where m is the size of the bucket space and n is the size of the key space. For $H = 5$, $n = 2^{32}$ and $m = 2^{12}$ we get $E[x] = 3.7 \times 10^{-9}$, which is exceedingly small. Thus solving the reverse sketch problem is

an effective way to converge to the set of heavy change keys. To reduce false negatives, in [23] we consider the more general version of this problem in which we are interested in finding the set of keys that map to heavy buckets in at least $H - r$ of the H hash tables. For simplicity in this paper we focus on the algorithms for the case of $r = 0$.

2.3 Architecture

The conceptual framework of our change detection system has two parts as in Fig. 1: streaming data recording and heavy change detection. Next, we will introduce each part in this system.

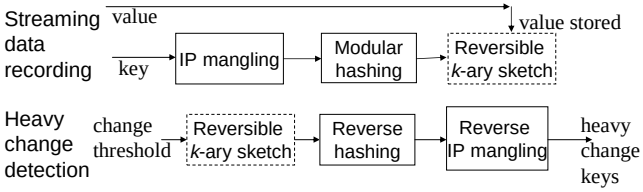

Figure 1: Conceptual architecture of the reversible k-ary sketch based heavy change detection system.

3. RECORDING OF DATA STREAMS

The first phase of change detection involves receiving ⟨key, update⟩ pairs one after another from an incoming data stream, and recording them in a summary data structure. As discussed in Section 1, each update should require very few memory accesses and the entire summary structure should be small enough to fit into fast memory. These requirements are fulfilled by the k-ary sketch. However, to allow reversibility, we modify the update procedure of the k-ary sketch with *modular hashing* (see Section 3.1). To maintain the accuracy of the sketch with this type of hashing, we also need to perform *IP-mangling* (see Section 3.2).

3.1 Modular Hashing

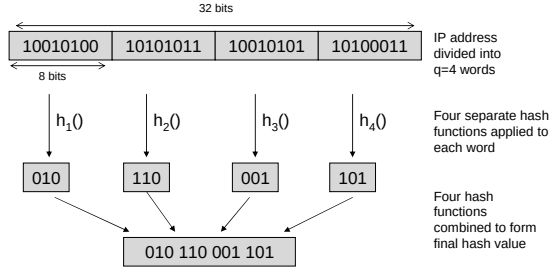

Figure 2: Modular hashing uses q hash functions to hash each word of the key, which are then combined for the final hash.

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 $O(q \cdot H)$. In Section 4, we discuss how this 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. Modular hashing does not have this property. Consider network traffic streams, which exhibit 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 is 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, the distribution of the number of keys per bucket was highly skewed, with most of the IP addresses going to a few buckets as shown in Figure 3. This significantly disrupts the estimation accuracy of our reversible k-ary sketch. To overcome this problem, we introduce the technique of *IP mangling*.

3.2 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.

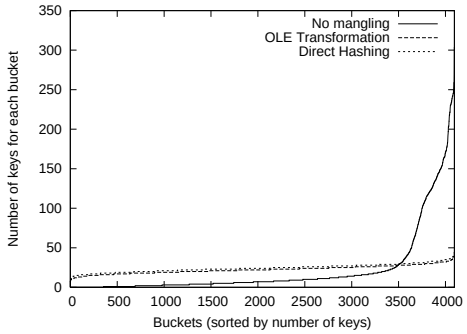

Figure 3: Distribution of number of keys for each bucket under three hashing methods.

The general framework for the technique is to use a bijective (one-to-one) 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.

Consider a function of the form $f(x) \equiv a \cdot x \mod n$. Such a function is invertible if and only if a and n are relatively prime. We refer to this as the OLE (odd linear equation) transformation. We are interested in values of n equal to 2^{32} in the case of an IP address. Thus for any odd $a \in [n]$ such a function on the domain $[n]$ yields a bijection. Our implementation is to choose uniformly at random an odd value for a from 1 to $n - 1$. The mangled key can easily be reversed by computing a^{-1} [1] and applying the same function to the mangled key, using a^{-1} instead of a. This scheme is good at randomly mapping keys independently as long as their suffixes differ. Ideally, n would be a prime and we could choose

any a from 0 to $n-1$ and thus have a universal hashing scheme. The weakness of our method can be seen in that for any two keys that share the last k bits, the mangled versions will also share the same last k bits. Thus distinct keys that have common suffixes will be more likely to collide than keys with distinct suffixes. However, in the particular case of IP addresses, this is not a problem. Due to the hierarchical nature of IP addresses, it is perfectly reasonable to assume that there is no correlation between the traffic of two IP addresses if they differ in their most significant bits. We thus believe that this mapping will sufficiently alter the original set of keys such that the locality (in terms of hamming distance [12] or absolute difference) of streaming keys will be destroyed.

We find that in practice our intuition holds true and the mangling 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 OLE transformation for IP mangling, and 3) direct hashing with 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 OLE 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. We note that for non-hierarchical keys, such as source/destintion pairs of IP addresses, an alternate (and slightly less efficient) scheme needs to be used. Such a scheme is described in [23].

Note that no extra memory or memory access is needed for modular hashing or IP mangling. Modular hashing of each word with small number of bits can be performed efficiently without pre-storing the mappings and then executing the table lookup. We can simply ignore any bits higher than $\log n$ for the modular operation of the OLE transformation.

4. REVERSE HASHING

4.1 Single Heavy Bucket

Once we have updated the sketch for each item in the data stream we want to obtain the set of suspect heavy change keys from the sketch, *i.e.*, the set of keys that hash to heavy buckets in each of the H hash tables. There are two possible approaches to solve this problem. The first is to iterate through all possible keys in the space $[n]$ and output the keys that hash to some heavy buckets for all the hash tables. This brute force approach obviously is not scalable.

The second basic approach is to perform bucket intersections. Suppose for each hash table i there is exactly one heavy bucket. Denote the set of keys that hash to the heavy bucket in table i as A_i. We can determine the set of suspect keys by computing the set $\bigcap_{i\in[H]} A_i$. This approach in general is also not scalable. Each set A_i is expected to be of size $\frac{n}{m}$. To perform detection, we need to obtain this set of keys for any given bucket in the sketch. This requires to store the mapping for the whole key space. In addition, it is inefficient to take intersections of such large sets, For example, for 32-bit IP address keys and $m = 2^{12}$, the sets A_i are each of expected size 2^{20}.

However, with modular hashing we can store and intersect these sets more efficiently. To implicity represent the sets A_i, for each bucket we store in memory q reverse lookup tables that map the bucket to its *modular bucket potential* sets $A_{i,1}, A_{i,2}, \ldots, A_{i,q}$. That is, if the index of the bucket corresponding to A_i is $y_1.y_2.y_3.y_4$ for $q = 4$, then a modular key $x_w \in [n^{\frac{1}{q}}]$ is in $A_{i,w}$ if $h_{i,w}(x_w) = y_w$. These modular potential sets give a compact representation of each 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, we can compute $\bigcap_{i\in[H]} A_i$ by computing $\bigcap_{i\in[H]} A_{i,w}$ for each w from 1 to q. That is, a key x is in $\bigcap_{i\in[H]} A_i$ if and only if the w^{th} word x_w of x is in $\bigcap_{i\in[H]} A_{i,w}$ for each w from 1 to q.

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 $H = 5$ 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 $O(q \cdot H \cdot (\frac{n}{m})^{\frac{1}{q}})$. Without modular hashing, the intersection takes $O(H \cdot \frac{n}{m})$. For the parameter values given above, our method yields $q \cdot H(\frac{n}{m})^{\frac{1}{q}} = 4 \cdot 5 \cdot 32 = 640$ versus $H \cdot \frac{n}{m} = 5 \cdot 2^{20} = 5242880$.

Finally, we note that while increasing q decreases the the run time of reverse hashing, there is a limit. The size of the space the modular hash functions map to, $m^{\frac{1}{q}}$, must be greater than 1. There is thus a tradeoff between the size of m, which effectively determines the size of the sketch, and the size of q, which determines the efficiency of reverse hashing. In [23] we discuss in detail this tradeoff and give reasons for choosing $q = \Theta(\log \log n)$.

4.2 Multiple Heavy Buckets

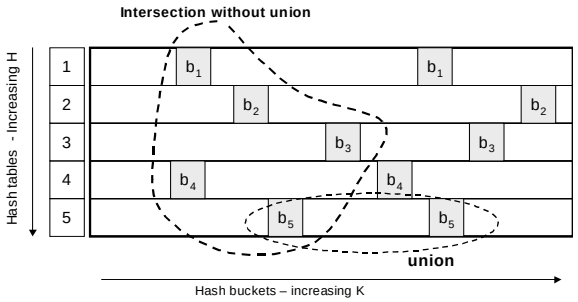

Figure 4: For two heavy changes, various possibilities exist for taking the intersection of each bucket's potential keys

We now consider how to generalize the method of reverse hashing described above to the case where there are multiple heavy buckets, say at most t, in each table. For $t \geq 2$ the problem is more difficult since it is not clear how to take the bucket intersections described for $t = 1$. For example, for $t = 2$ there are $t^H = 2^H$ possible ways to take the H-wise intersections discussed above as shown in Figure 4. One heuristic solution is to union all of the bucket potential sets in each hash table and intersect these H union sets. But it is easy to see that such a set can contain keys that do not hash to heavy buckets in each of the H hash tables. We thus expect to get a large number of false positives from this method. We could verify each output key of this method by estimating its value through k-ary sketch. But in our evaluation section we show that the number of false positives generated by this method grows exponentially in the number of heavy buckets t. Thus this verification procedure is applicable only for small values of t.

Our more efficient scheme is as follows. For each w from 1 to q we compute a set I_w which consists of the set of all $x_w \in [n^{\frac{1}{q}}]$ such that for each hash table i, there is some heavy bucket A such that x_w is contained in the modular potential set $A_{i,w}$ for A. In

addition, we attach to each element in I_w an H-dimensional *bucket vector* which denotes which heavy bucket in each hash table the corresponding modular word occurs in. Since a given modular key can potentially occur in up to t heavy buckets for a given hash table, each modular key for a word w can have multiple vectors. The set I_w may thus have multiple occurrences of a given key, once for each of its vectors. We then create a graph whose vertices are the elements of the q sets I_w. Edges are drawn between vertices $x \in I_{w-1}$ and $y \in I_w$ if the bucket vectors for elements x and y are the same. It then follows that any length q path through this graph corresponds to a key that hashes to heavy buckets in all H hash tables.

For example, in Figure 5, suppose $t = 5$ and each heavy bucket in each hash table is indexed from 1 to 5. From the figure we have that for key $a \in [n^{\frac{1}{q}}]$ the modular hash functions are such that $h_{0,1}(a) = 2$, $h_{1,1}(a) = 1$, $h_{2,1}(a) = 4$, $h_{3,1}(a) = 1$, and $h_{4,1} = 3$. We also have, by coincidence, that $f, g \in [n^{\frac{1}{q}}]$ hash to exactly the same values for each of the H hash functions $h_{i,3}$. Thus there are two length q paths through the graph, $a.d.f.i$ and $a.d.g.i$. These are thus the two suspect heavy change keys for the sketch.

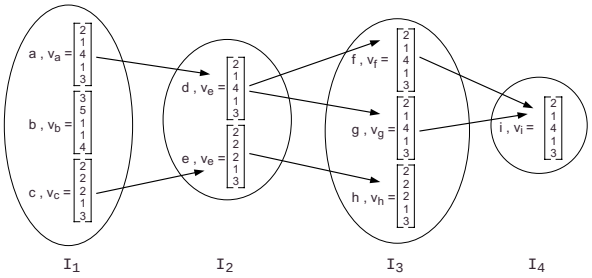

Figure 5: The sets I_w form a graph whose length q paths correspond to the heavy change keys.

4.3 Work in Progress

In [23], we give the details of our reverse hashing algorithms for multiple heavy changes, as well as various optimizations for reducing false positives and false negatives and for improving efficiency. We highlight those techniques as the following.

- We generalize the algorithm so that it detects keys that occur in heavy buckets for at least $H - r$ of the H hash tables. We can then adjust the parameter r to balance the trade-off between false positives and false negatives.
- We introduce *bucket index matrix* algorithm to significantly reduce the size of the produced graph. This allows us to detect larger numbers of heavy changes efficiently.
- To further increase the number of heavy changes we can efficiently detect, we propose the *iterative approach* to reverse hashing. This scheme can also help reduce false positives.
- To reduce false positives we use a second verifier sketch that uses 4-universal hash functions. We give analytical bounds on the false positives for this scheme.
- We introduce a new IP-mangling scheme with better statistical properties that permits reverse hashing with non-hierarchical keys such as source/destination IP address pairs.

5. PRELIMINARY EVALUATION

In this section we show some preliminary evaluation results of our schemes, and refer the readers to [23] for more comprehensive testing and results. Our evaluation is based on one-day *netflow* traffic traces collected from a large edge router in Northwestern University. The traces are divided into five-minute intervals with the traffic size for each interval averaging about 7.5GB. In addition to the reversible k-ary sketch, we implemented a naive algorithm to record the per-source-IP volumes, and find the top IPs with heavy changes as the ground truth. Then we use the volume change of the top x-th ($x = 20, 40, etc.$) IP as threshold to infer the top x IPs with heavy volume changes over that threshold through reversible k-ary sketch. Our metrics include *speed, real positive* and *false positive percentages*. 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.

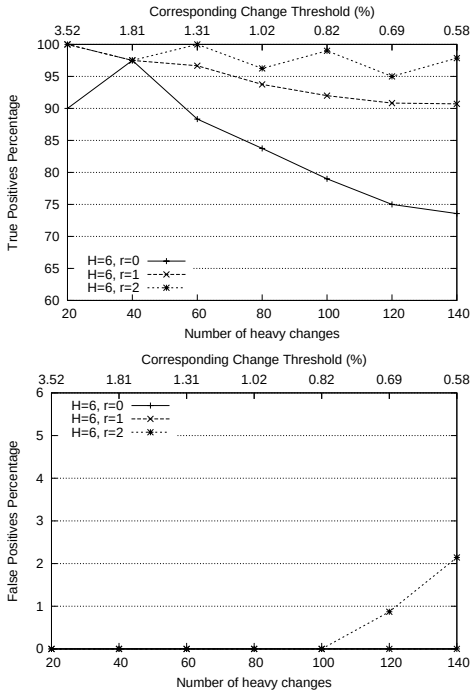

Figure 6: Heavy change detection accuracy: true positive (top) and false positive (bottom). The top-x axis show the corresponding change threshold ϕ defined in Section 2.2.

For single heavy change, we always have 100% real positive and zero false positive. For multiple heavy changes, we also achieve very high accuracy with the algorithms outlined in Sections 4.2 and 4.3. Figure 6 shows some sample results from [23], and compare them with those of the simple case $r = 0$. Here $H = 6$ and $k = 2^{12}$, thus only 192KB memory are used for recording the reversible k-ary sketches. The results are very accurate for $r = 2$: over 95% true positive rates for up to 140 heavy changes and negligible false positive rates; while the true positive rate drops significantly with $r = 0$ because it is very sensitive to the collision. In general, higher values of r (with $r < H/2$) results in higher true positive percentage with a slight degradation in false positive percentage. We tried several different traces from different time intervals and obtained similar results. Note that for any given key, k-ary sketch can only estimate its value with bounded errors. In our experiments, we found that all the heavy changes missed are due to boundary effects caused by estimation error, and all the major changes are detected.

Next, we compare our bucket-vector algorithm with the naive

way of intersecting the unions of buckets in each hash table as described earlier in Section 4.2. Figure 7 shows the number of false positives for the naive method for which the number of false positives rises exponentially even for a small number of true heavy changes. As described in more detail in [23], the reverse hashing process is very fast, taking only 0.42 seconds for up to 100 heavy changes with an un-optimized implementation on a Pentium-IV 2.4GHz PC.

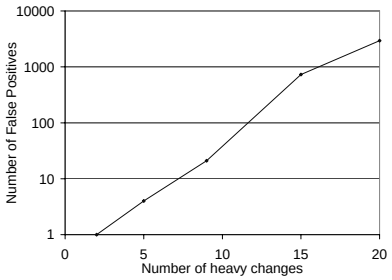

Figure 7: The number of false positives by intersecting the unions of buckets

6. RELATED WORK

For high-speed network monitoring, most existing high-speed network monitoring systems estimate the flow-level traffic through packet sampling [6, 7], but this has two shortcomings. First, sampling is still not scalable, especially after aggregation; there are up to 2^{64} flows even defined only by source and destination IP addresses. Second, long-lived traffic flows, increasingly prevalent for peer-to-peer applications, will be split up if the time between sampled packets exceeds the flow timeout [6].

Applications of sketches in the data streaming community have been researched quite extensively in the past. This has also been motivated by the emerging popularity of applications for network traffic accounting, anomaly detection and very large databases with massive data streams. Usually the work has focused on extracting certain data aggregation functions with the use of sketches, like quantiles and heavy hitters [4, 3, 11, 2], distinct items [9] etc.

Recently, Cormode and Muthukrishnan proposed *deltoids* approach for heavy change detection [5]. 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. For instance, it requires more than 1MB to detect 100 or more changes, and therefore cannot even fit into the latest FPGAs, which only has up to 600KB of block SRAM that can be efficiently utilized [27].

7. CONCLUSIONS

We have proposed novel reverse hashing methods for improving sketch-based change detection in high speed traffic. Our techniques can efficiently and accurately output the set of keys which show heavy change in two different time intervals, without storing the key information explicitly. Being able to *reverse* a sketch in this fashion is a key step to enhance the power of sketch based change detection to online, single pass settings. Without adding any memory or memory access to record the streaming data, our algorithms use sub-linear time and space in the size of the key space for heavy change detection [23], and are scalable to large key spaces.

8. REFERENCES

[1] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1989.

[2] CORMODE, G., ET AL. Finding hierarchical heavy hitters in data streams. In *Proc. of VLDB* (2003).

[3] CORMODE, G., ET AL. Holistic UDAFs at streaming speeds. In *Proc. of ACM SIGMOD* (2004).

[4] CORMODE, G., AND MUTHUKRISHNAN, S. Improved data stream summaries: The count-min sketch and its applications. Tech. Rep. 2003-20, DIMACS, 2003.

[5] CORMODE, G., AND MUTHUKRISHNAN, S. What's new: Finding significant differences in network data streams. In *Proc. of IEEE Infocom* (2004).

[6] DUFFIELD, N., LUND, C., AND THORUP, M. Properties and prediction of flow statistics from sampled packet streams. In *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMW)* (2002).

[7] DUFFIELD, N., LUND, C., AND THORUP, M. Flow sampling under hard resource constraints. In *Proc. of ACM SIGMETRICS* (2004).

[8] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting. In *Proc. of ACM SIGCOMM* (2002).

[9] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (1985), 182–209.

[10] GILBERT, A. C., ET AL. QuickSAND: Quick summary and analysis of network data. Tech. Rep. 2001-43, DIMACS, 2001.

[11] GILBERT, A. C., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. J. QuickSAND: Quick summary and analysis of network data. Tech. Rep. 2001-43, DIMACS, 2001.

[12] HAMMING, R. W. *Coding and Information Theory*. Prentice Hall, 1986.

[13] HOFMEYR, S., AND FORREST, S. Intrusion detection using sequences of system calls. *Journal of Computer Security 6* (1998).

[14] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. of ACM SIGCOMM IMC* (2003).

[15] LOYALL, J. P., ET AL. Building adaptive and agile applications using intrusion detection and response. In *Proc. of NDSS* (2000).

[16] MANKU, G. S., AND MOTWANI, R. Approximate frequency counts over data streams. In *Proc. of IEEE VLDB* (2002).

[17] MIRKOVIC, J., AND REIHER, P. Data streams: algorithms and applications. In *Proc. of ACM SODA* (2003).

[18] MOORE, D., ET AL. The spread of the Sapphire/Slammer worm. *IEEE Magazine on Security and Privacy* (August 2003).

[19] MUTHUKRISHNAN. Data streams: Algorithms and applications (short). In *Proc. of ACM SODA* (2003).

[20] PAXSON, V. Bro: A system for detecting network intruders in real-time. *Computer Networks 31*, 23-24 (1999), 2435–2463.

[21] ROESCH, M. Snort: The lightweight network intrusion detection system, 2001. `http://www.snort.org/`.

[22] RYUTOV, T., NEUMAN, C., KIM, D., AND ZHOU, L. Integrated access control and intrusion detection for web servers. *IEEE Trans. on Parallel and Distributed Systems 14*, 9 (2003), 841–850.

[23] SCHWELLER, R., CHEN, Y., PARSONS, E., GUPTA, A., MEMIK, G., AND ZHANG, Y. Reverse hashing for sketch-based change detection on high-speed networks. Tech. Rep. NWU-CS-2004-45, Northwestern University, 2004.

[24] STANIFORD, S., PAXSON, V., AND WEAVER, N. How to own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium* (2002).

[25] WAGNER, D., AND DEAN, D. Intrusion detection via static analysis. In *Proc. of the IEEE Symposium on Security and Privacy* (2001).

[26] WEAVER, N., PAXSON, V., STANIFORD, S., AND CUNNINGHAM, R. Large scale malicious code: A research agenda. Tech. Rep. DARPA-sponsored report, 2003.

[27] XILINX INC. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet, 2004. `www.xilinx.com/bvdocs/publications/ds083.pdf`.