

# Detecting Attacks in Routers Using Sketches

Dhiman Barman, Piyush Satapathy, Gianfranco Ciardo  
Department of Computer Science and Engineering  
University of California, Riverside

**Abstract**— Designing routers against different attacks is imperative in today’s Internet. We propose accurate, memory efficient, and scalable techniques to detect attacks such as worms, viruses, superspreaders, and denials-of-service (DoS) in routers. Our schemes enable detection in the routers by looking only at the IP headers. We propose a general methodology to use sketches, in particular count-min sketch, FM sketch, and counting and multi-counting Bloom filters, to recognize attacks in the routing architecture. Our techniques are based on change detection, for which we propose an algorithm that can work on data-streams and leverage the accurate and efficient estimation provided by sketches. We evaluate the performance of different schemes on real traces to show their accuracy.

## I. INTRODUCTION

With the proliferation of various kinds of attacks, designing secure and fault-resilient communication systems has become an indispensable network design goal. Resources consumed and performance degradation caused by worms and viruses are significant at the end-user applications. Recent worm attacks (e.g., Code Red version 1 and 2, nimba) have shown how easy it is for a worm to spread and gain control of hundreds of computers in a few minutes and attack a target bottleneck. Recently, low-rate attacks [16] have been discussed in the literature; these are capable of causing considerable TCP throughput loss by injecting small amounts of traffic at the right intervals. Deployment of such protocols leaves the door wide open for resource abuse and security threats [21].

State-of-the art techniques are often based on Principal Component Analysis (PCA) [17] to detect anomalies in traces collected from multiple locations over same window of time, mostly off-line. Data-mining and time-series analysis have been used for off-line trend analysis in the network traffic [14] under certain traffic models. Moreover, association rules have been used to detect resource consumption in the context of network traffic [7]. Most current techniques focusing on detecting anomalies off-line cater to the needs of data engineering at long time scales and help ISP’s capacity planning. However, end-users and applications are more concerned about performance guarantees at much shorter time scales. Therefore, steps should be taken to detect and filter attacks at these shorter time scales, to improve user-perceived performance. Recent proposals advocate routers with more capabilities to detect malicious flows [6], [12]. We are faced with many other demands for user-defined aggregate functions [4] and their support in routers for real-time processing and query answering. Changes in this direction are being put in practice. Cisco’s Traffic Anomaly Detector XT works off the critical path, monitoring mirrored traffic flows at gigabit line rates to build profiles of “normal” behavior for each protected device.

[10] describes a space-efficient simple data structure using Bloom filters to represent a static data set. [15] presents a lossy data structure using counting sketches to capture the flows in the network. A closely related work [3] suggests a method for detecting changes of the “heavy” flows using a sequential hashing scheme which is efficient in memory and overhead. However, the approach in [3] is different from our sketch-based change detection in that we use sketches to build summaries and use different algorithms to find changes among those summaries. In this paper, we adopt these ideas and use them to derive efficient methods that detect attacks on-line in routers. A reasonable way to detect malicious activity at the router level is for the router to compare the data flowing through it with *what the router would expect to be flowing through it*. Such a scheme implicitly assumes that the attack has some sort of signature that distinguishes it from normal traffic. For such a scheme to work on-line, the router must be able to create a compact representation of a large volume of data in real-time. We believe that sketches [5] can provide a good trade-off between the computation required to build a summary and its accuracy. We provide both primitives for queries and various approaches that use sketches to detect attacks. Once a set of flows has been identified as malicious, several approaches can then be used to contain the possible damage, such as isolating those flows to a different queue.

The paper is organized as follows. Section II discusses simplified models of attacks. Section III provides background on sketching and details our sketching technique to detect attacks on-line. Section IV presents our main contributions, proposing algorithms that use sketches for change detection. Section V evaluates our schemes on real Internet data. We conclude in Section VI.

## II. DEFINITIONS AND ATTACKS

Today, the malicious activities carried out by worms, viruses, scans, and malwares originated from botnets are prevalent in the Internet. To design space and time efficient algorithms that can detect such attacks, we first need to understand and characterize the nature of these attacks.

A data-stream  $S$  is a sequence of tuples  $\langle (i, j, p, q), v \rangle$ , where  $i$  and  $j$  denote the packet source IP address and port,  $p$  and  $q$  denote the destination IP address and port, while  $v$  is a value (e.g., size) associated with the packet. A flow (as given by Cisco’s NetFlow)  $R(i, j, p, q)$  is the multiset containing all the packets corresponding to a given  $\langle i, j, p, q \rangle$  combination.

We can then characterize various attacks as follows. Port-scans are attacks where a particular IP address and port pair

connects to a destination on several ports:

$$\text{PortScan}(i, j, p) \Leftrightarrow |\{q : |R(i, j, p, q)| > 0\}| > \delta_{PS},$$

where  $\delta_{PS}$  are a user-defined threshold. Address-scans are attacks where a particular IP address connects to multiple destination IP addresses on a particularly vulnerable port. The source host may or may not use different source ports. Such attacks can be identified by a large number of flows destined to a given destination port:

$$\begin{aligned} \text{AddrScan}(i, j, q) &\Leftrightarrow |\{p : |R(i, j, p, q)| > 0\}| > \delta_{AS} \quad \text{or} \\ \text{AddrScan}(i, q) &\Leftrightarrow |\{(j, p) : |R(i, j, p, q)| > 0\}| > \delta_{AS}, \end{aligned}$$

where  $\delta_{AS}$  is user-defined threshold.

We consider simple models of spam, malware, and worms. *Spams* are usually sent from a server (or set of botnets) to a large set of destinations, although this is also typical behavior for peer-to-peer clients or content-providers. *Worms* and *malware* spread like epidemic: initially, a set of hosts sends malicious packets to other hosts after scanning; then, the victims further spread malicious contents, recursively. The assumption is that few sources try to connect to a particular destination or a set of destinations on any ports.

$$\text{WormMalwSpam}(i) \Leftrightarrow |\{(j, p, q) : |R(i, j, p, q)| > 0\}| > \delta_{WMS},$$

where  $\delta_{WMS}$  is user-defined threshold.

Flows are termed ‘‘heavy hitters’’ if they send most of the packets or bytes, so we can modify the above formulations to incorporate heavy-hitters. Worms are sometimes termed *superspreaders*.

There are other kinds of flows which can be flagged as suspicious such as *DoS* marked by sudden increase in the number of packets; *alpha flows*, by sudden increase in the number of packets and bytes per packet; *flash crowd*, by sudden increase in the number of source IP addresses, source ports, number of packets and bytes per packet.

### III. BACKGROUND

We first discuss background on sketches, then use them as a lossy data structure in our technique to analyze attack traffic. For sketches to be useful in the detection mechanism, the flow ids seen by the routers cannot be manipulated by the attackers. Therefore, we assume the adoption of a technique to prevent spoofing of source and destination IP addresses.

With the increasing speed of Internet links, traffic monitoring is becoming increasingly challenging. For example, on a 40Gbps link, a router has about 25ns to forward the packet, so any additional packet processing has severe time constraints. A data-stream sketch is a compact summarization much smaller than the data-stream itself. For a data-stream over a domain of size  $N$ , the sketch is of size  $\log^{O(1)}(N)$ . Sketches can be computed very fast and can be collected at distributed points. Furthermore, sketches can accurately answer queries about the data-stream most of the time.

‘‘Tug-of-war’’ sketches [1] are computed using 4-wise random hash functions  $g_j$  that map packets to  $\{+1, -1\}$ . Let  $a_i$  be the number of occurrences of an object  $i \in \{1, \dots, N\}$  in a data-stream  $a$ . A sketch with a relative error no greater

than  $\epsilon$  and with at least  $\delta$  confidence is a vector of length  $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ , whose  $j^{\text{th}}$  entry is defined to be  $\sum_{i=1}^N a_i g_j(i)$ , and is easy to maintain under updates. This structure has been applied to find the (approximate) inner product of two vectors  $a$  and  $b$  with error  $\pm \epsilon \|a\|_2 \|b\|_2$  [9]. To keep up with the link speeds, a small number of sketches can be stored in SRAM. As new packets arrive, the sketch is updated. A sketch has the property that we can generate linear projections (inner products) of the data-streams with a small (polynomial) number of vectors quite easily and accurately, provided the dot product of corresponding unit vectors (the *cosine*) is large. This can be used in several ways. First, since any point query  $i$  on the signal can be viewed as merely the inner product of the signal with a vector that has a 1 in its  $i^{\text{th}}$  component and 0 elsewhere, we can use the sketch to directly estimate the point query; likewise for range queries. Since there are only  $N$  point queries and  $N(N-1)/2$  range queries, which are small polynomials, sketches will suffice. Second, since wavelet transforms are linear projections of the signal with a specific set of  $N$  vectors, we can generate wavelet coefficient approximations from the sketch [9] which can in turn be used for point or range query estimations on the signal up to error of  $\pm \epsilon \|a\|_2$ .

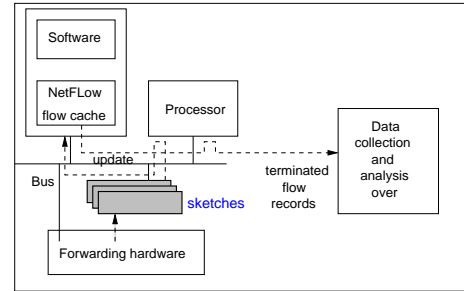


Fig. 1. Sketches in router linecard.

**Count-Min Sketch.** Next we discuss about Count-Min (CM) sketches [5]. CM sketches maintain an  $n$ -dimensional vector which is updated upon arrival of each packet (and its attributes). The current state at time  $t$  is  $\mathbf{a}(t) = \langle a_1(t), \dots, a_n(t) \rangle$ . Initially  $\mathbf{a}$  is the zero vector  $\mathbf{0}$ , so  $a_i(0) = 0$  for all  $i$ . Updates to the individual entries of the vector are presented as a stream of pairs. The  $t^{\text{th}}$  update is  $(i_t, c_t)$  and the vector is updated as  $a_{i_t}(t) = a_{i_t}(t-1) + c_t$  and  $a_{i'}(t) = a_{i'}(t-1)$  for  $i' \neq i_t$ .

Count-Min sketch is defined with two parameters  $(\epsilon, \delta)$  and can be stored in an array of width  $w$  and depth  $d$ . Given parameters  $(\epsilon, \delta)$ , let  $w = \lceil \frac{1}{\epsilon} \rceil$  and  $d = \lceil \ln \frac{1}{\delta} \rceil$ . We also need a set of hash functions  $h_1, \dots, h_d : \{1, \dots, n\} \rightarrow \{1, \dots, w\}$  which are chosen uniformly at random from a pairwise-independent family. To update the sketch upon arrival of each element, we have  $\text{count}[j, h_j(i_t)] \leftarrow \text{count}[j, h_j(i_t)] + c_t$ , for  $1 \leq j \leq d$ .

At any time  $t$ , monitoring tasks might invoke queries to compute certain functions of interest on  $\mathbf{a}(t)$ . Effectiveness of sketching techniques will be realized, if a router supports query primitives to answer queries regarding plausible attacks

based on vectors  $\mathbf{a}$  and  $\mathbf{b}$  corresponding to two data-streams. Such primitives will be:

- a *point query*, denoted by  $\mathcal{Q}(i)$ , to return an approximation of  $a_i$  given by  $\min_j \text{count}[j, h_j(i)]$
- a *range query*  $\mathcal{Q}(l, r)$  to return an approximation of  $\sum_{i=l}^r a_i$
- an *inner product query*, denoted by  $\mathcal{Q}(\mathbf{a}, \mathbf{b})$  is to approximate  $\sum_{i=l}^r a_i b_i$  given by  $\sum_{k=1}^w \text{count}_a[j, k] \cdot \text{count}_b[j, k]$ .
- a *change range query*, denoted by  $\mathcal{Q}(l, r, e_1, e_2)$  is to return an approximation of  $\sum_{i=l}^r a_i$  between events  $e_1$  and  $e_2$  and we discuss it later.

**Bloom filter.** A Bloom filter [10] is a simple space-efficient randomized data structure to represent a multiset of  $n$  elements and support membership queries. When a list or set is used and space-usage needs to be minimized, a Bloom filter is often a good choice. A Bloom filter is an array of  $m$  bits initially all bits are set to 0. An incoming element with identifier  $id$  is hashed through  $k$  hash functions and the bit positions returned by the hash functions are set to 1. A location can be set to 1 multiple times but only the first change has an effect. Hence a Bloom filter may yield a false positive with probability [2]  $(1 - \rho)^k$ , where  $\rho = e^{-kn/m}$ . In a counting Bloom filter, each bit in the Bloom filter is replaced by a small counter. When an item is inserted,  $k$  hash functions map the item to  $k$  buckets and the corresponding counters are incremented by the value associated with the  $id$  and when an item is deleted the corresponding  $k$  counters are decremented. Considering  $n$  elements,  $m$  counters, and  $k$  hash functions,  $O(\log \log m)$  bits are necessary for each counter to avoid overflow. A multi-counting Bloom filter is a counting Bloom filter with  $m$  counters and  $m$  buckets divided into  $k$  groups of size  $m/k$ . The range of the  $i^{\text{th}}$  hash function maps to buckets  $\{m(i-1)/k + 1, \dots, mi/k\}$ .

**FM sketch.** An FM sketch [8] is an unbiased estimator of the number  $N$  of distinct elements in a multiset and is order- and duplicate-insensitive. An FM sketch assumes a pseudo-random hash function which maps elements from  $[0 \dots 2^L] \rightarrow [0 \dots L]$  into a bitmap. Let  $\rho(x)$  represent the position of the least significant 1-bit in the binary representation of  $x$  and  $\rho(0) = L$ . If the hash values are uniformly distributed, the position of the least significant 1-bit in the bitmap is  $k$  with probability  $\frac{1}{2^{k+1}}$ . The duplicate insensitivity arises from the fact that a bitmap hash function maps an element to a fixed bit every time it is inserted. The leftmost zero position,  $R$ , is used as an indicator of  $\log_2 N$ . The expectation of  $R$  is

$$E[R] \approx \log_2(0.773N) \quad \text{which implies} \quad N \approx 1.29 \times 2^{E[R]}.$$

Here  $E[R]$  is an unbiased estimator but the variance of  $R$  is 1.12. To improve the variance [8] proposes to use an average over  $m$  independent bitmaps (Probabilistic Counting with Stochastic Averaging, PCSA). To achieve a  $O(1)$  time complexity, PCSA maintains  $m$  independent bitmaps and each element is mapped to one bitmap using another hash function. Thus, on average,  $N/m$  distinct elements are mapped to a bitmap. The modified estimator is thus  $N \approx 1.29m2^{E[\sum k_i/m]}$  where  $k_i$  is the least significant bit position of the 1-bit in the

$i$  the FM sketch. The relative error is  $0.78/\sqrt{m}$ . Moreover, as shown in the analysis,  $O(\log_2 N)$  should be an appropriate value of the sketch sizes for  $N$  distinct elements with space complexity  $O(m \log_2 N)$ .

#### IV. OUR CONTRIBUTIONS

Having described sketches, we now propose several change detection algorithms based on them.

- In our first approach, to capture unexpectedness, we define the change between two sketches  $S_1(\alpha_1, \beta_1)$  and  $S_2(\alpha_2, \beta_2)$  as  $S_d = S_2 - kS_1$ , since sketches are closed under linear combination;  $\alpha_i$  and  $\beta_i$  denote the beginning and end of the  $i^{\text{th}}$  monitoring interval and  $k$  is a threshold on the tolerable relative change. To give an example, assume that portscans activity is prevalent during an interval and is reflected through a high scanning rate. An alternative approach could be to maintain two sketches,  $S_1$  and  $S_2$  such that  $S_1$  is updated when the scanning rate is high (or congestion is high) and  $S_2$  is updated when there is no scanning. Using simple queries, accurate detection of the flows showing abrupt changes can be achieved.
- In the second approach, interactive *periodic abrupt change* discovery involves tasks; first we have to detect the periodic abrupt changes, then we need to *age* the summarization in sketches so that the router can pose queries. Our approach to discover attacks in a traffic is based on the computation of the moving average (MA) of the sketches, with a subsequent annotation of abrupt changes as the points with value higher than  $x$  standard deviations about the mean value (of point query) of the MA. The width of the moving window affects the sensitivity and accuracy of the estimation. We can use estimators as in [14] for mean and variance.

---

1: Maintain  $w$  sketches,  $S = \{s_1, s_2, \dots, s_w\}$ , corresponding to the last  $w$  measurement intervals

2: Update  $s_{MA} = \sum_{i=1}^w g_i s_i / \sum_{i=1}^w g_i$ ,  $g_i$  is weight of  $s_i$

3: Compute  $\bar{x}_i = \text{median}_j v_x^{h_i}$  where

$$\bar{v}_x^{h_i} = \text{count}[i, h_i(x)] - \text{sum}(S)/K/1 - 1/K$$

$$\text{sum}(S) = \sum_j \text{count}[0][j]$$

4: Compute  $\text{var} = \text{median}_j \text{var}^{h_i}$  where

$$\text{var}_x^{h_i} = \frac{K}{K-1} \sum_j (\text{count}[i][j])^2 - \frac{1}{K-1} (\text{sum}(S))^2$$

5: Attacks =  $\{f_i | \text{flow ids from buckets in } s_w \text{ that exceed } \text{mean}_i(\bar{x}) + \beta \text{var}\}$  where  $\beta$  is user-defined constant factor.

---

##### A. Change Detection based on Probability Distribution

Any attack detection algorithm should be based on a change detection algorithm. Our algorithm is similar to that proposed in [13] but our focus here is to emphasize the accuracy and

compactness of sketches in a change-detection algorithm. The main idea behind Algorithm 1 is to detect the points where the underlying distribution of the data stream elements, captured using sketches, change. We present the case using FM sketches that estimate the number of distinct flows in regular intervals, where flows are identified by 5-tuples. As assumed before, this technique of detecting drastic changes in the number of distinct flows will give an indication of scanning activity – portscan, address, or other attacks. Another advantage behind leveraging the meta-algorithm in [13] is that it provides theoretical bounds on misclassification.

---

**Algorithm 1** Change detection using FM sketches

---

```

1:  $c_0 \leftarrow 0$ 
2: for  $i = 1 \dots k$  do
3:    $s_i \leftarrow FM_i$ 
4:    $wind_{X,i} \leftarrow m_{X,i}$  intervals from time  $c_0$ 
5:    $wind_{Y,i} \leftarrow$  next  $m_{Y,i}$  intervals in incoming data streams
6: end for
7: while more flows to process do
8:   Slide  $wind_{Y,i}$  by 1 sample
9:   if  $distance(wind_{X,i}, wind_{Y,i}) \geq \alpha_i$  then
10:     $c_0 \leftarrow$  current time
11:    Output change at time  $c_0$ 
12:    Clear all windows and goto step 1
13:   end if
14: end while

```

---

In Algorithm 1, we keep  $k$  pairs of windows (sliding in parallel). Each window spans over multiple intervals and each interval has a count estimated by a FM sketch. In the process of comparing  $k$  pair of windows (one is a base window), if the distance exceeds a threshold, we declare a change. Another important issue is the distance function, for which we propose several possibilities: (a) a modification of the distance function from [13] and (b) the difference of the sum of values between two windows (denote by sum-diff). One can also use the KL-distance function<sup>1</sup>. We describe a distance function in Algorithm 2 that is used by Algorithm 1.

---

**Algorithm 2** Calculating distance between two Windows

---

**Input:**  $S_1$  and  $S_2$  are two windows of any lengths  
**Output:**  $d$ , statistical distance between  $S_1$  and  $S_2$

```

1:  $j \leftarrow MaxFl / \max(|S_1|, |S_2|)$  { $MaxFl$  is a threshold}
2:  $\mathcal{A} \leftarrow \{A_i | A_i = [a_i, b_i], a_i < b_i, (a_i, b_i) \sim U[0, MaxFl], i = 1, \dots, j\}$ 
3: for  $i = 1 \dots j$  do
4:    $S_1(A_i) \leftarrow |S_1 \cap A_i| / |S_1|$ ,  $S_2(A_i) \leftarrow |S_2 \cap A_i| / |S_2|$ 
5:    $dist_i \leftarrow 2|S_1(A_i) - S_2(A_i)|$ 
6: end for
7:  $d \leftarrow \max_j dist_j$ 

```

---

<sup>1</sup>Given two windows  $W_1$  and  $W_2$  and their associated multiset of flow counts  $s_i$  in a data stream  $\{s_1, s_2, \dots, s_n\}$ , the KL distance from  $W_1$  to  $W_2$  is given by  $D(W_1 || W_2) = \sum_{a \in A} P_{w_1}(a) \log \frac{P_{w_1}(a)}{P_{w_2}(a)}$ , where the empirical distribution is given by  $P_w(a) = \frac{N(a|w)+0.5}{n+|A|/2}$  and  $A$  is an interval of the form  $[a, b]$ ,  $a, b \in \mathbb{Z}^+$ .

Intuitively, Algorithm 2 tries to find the largest change in probability of a set. Given two finite domain subsets,  $S_1$  and  $S_2$  and a collection of measurable subsets denoted by  $A_i$ s,  $d$  returns the largest variation distance between two distributions represented by  $S_1$  and  $S_2$ . Line 4 gives the empirical weight of  $A_i$  w.r.t to  $S_j$  and empirical distance is given by Line 5. We construct the measurable subsets by taking intervals from the real line in Line 2.

**Other issues.**

Sketches are inherently irreversible and are good for summarizing and querying only if we know the flow ids. It is difficult to obtain the flow ids corresponding to *heavy-hitters* or *superspreaders*. Once we detect the heavy change buckets, the main problem is retrieving the culprit keys from the Count-Min sketch. Some techniques have been proposed for offline detection [20]; some detect flow ids by searching in small hierarchical key space [3]. In case of a single bucket change in each hash table, we can recover the culprit key with some accuracy if we simply take the intersection of the reverse mappings for the heavy change buckets in each hash table. For our sketching schemes to work, we assume absence of spoofing. In [11], the authors proposed a new network architecture to overcome this problem: the IP address space is divided into a set of client addresses and a set of server addresses. This allow clients to initiate connections to servers, but does not allow clients to initiate connections to clients, nor servers to initiate connections to servers. With this architecture, many DoS and reflection DoS can be prevented. Moreover, the client address does not need to have a global significance, it just needs to have significance along the path between the client and the server, so that packets from the server can return to the client (this does not have any effect on the sketch mechanism, as all is needed is a unique identifier). Path-based client addresses make complete source-address spoofing impossible for the clients.

## V. EVALUATION

We consider three different sketching techniques, count-min (with  $\epsilon = 0.01$  and  $\delta = 0.1$ ), counting Bloom filter, and multi-counting Bloom filter. To avoid counter overflow we choose 1000 counters and four hash functions. For the third sketching technique we implement a parallel multi-stage Bloom filter. We have a set of  $m$  counters and we divide those into  $k$  subsets of  $m/k$  counters per set using  $k$  independent hash functions. Every insertion increments one counter in each one of the  $k$  subsets. Again we use 1000 counters and four hash functions.

For change detection we implement an FM sketch with 100 bits and two bit-vectors. We consider two different traces; a general traffic flow [18] that contains few unique flows and a malicious traffic flow (Mydoom trace [19]) that contains thousands of unique flows. We partition each trace into ten uniform time intervals and calculate the distinct flows for each part of the trace by averaging the values over ten different runs with different hash functions.

First, we provide some evaluation results that supports our claims about using sketching techniques. Due to lack of space, we only present representative results that validate Algorithm 1

and the first approach in Section IV. Fig. 2(a) compares the flow size using three different sketching techniques and the exact value. We plot ten heavy flows out of 436 total flows from a trace file and observe that the estimated flow sizes are quite accurate. Fig. 2(b) represents the accuracy of change in flow sizes between two time intervals. We divide the traces into two time intervals and calculate the changes in flow sizes across the two intervals for each flow. Comparison of the estimated changes against exact changes shows that estimation is quite accurate. Fig. 3(a) shows the FM sketch estimation over ten intervals of the general traffic trace and Fig. 3(b) shows the estimation over ten intervals of the Mydoom trace. In both figures, we see small errors in estimation for the FM sketch (we know that the FM sketch estimate has some variance). Here we intend to show that the FM sketch gives a quite good approximation of the exact number of distinct flows, however the error percentage could go higher in certain cases due to the higher covariance nature of the hashing functions used in the FM sketch.

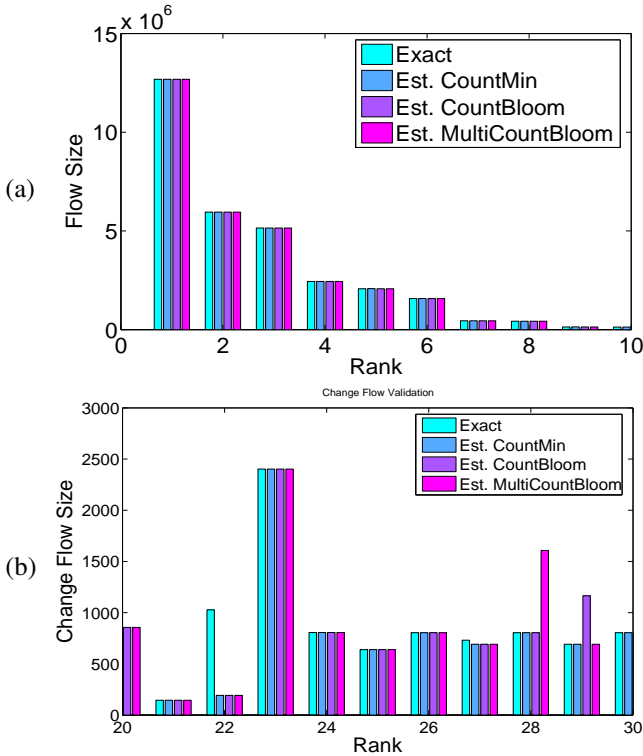


Fig. 2. (a) Sketches of ten heavy ranked flows: A comparison of the exact flow size with the estimated values using three different sketching techniques. (b) Sketching changes in heavy ranked flows between two time intervals: A comparison of change in flows over two periods with that of three different sketching techniques over the same period. We count flow sizes in bytes.

Next, we evaluate our proposed sketching techniques using a malicious trace of 100K packets. We partition the trace into 100 parts and for every part we estimate the FM sketch using Algorithm 1 and detect changes using the distance algorithm of Algorithm 2. We also calculate the change detection using the exact value of the flow counts and using the sum-diff distance algorithm. In Fig. 4(a), we plot the normalized error between the exact change detection and that estimated by the FM sketch. We calculate the error as follows: Let  $V_{fm}$  be

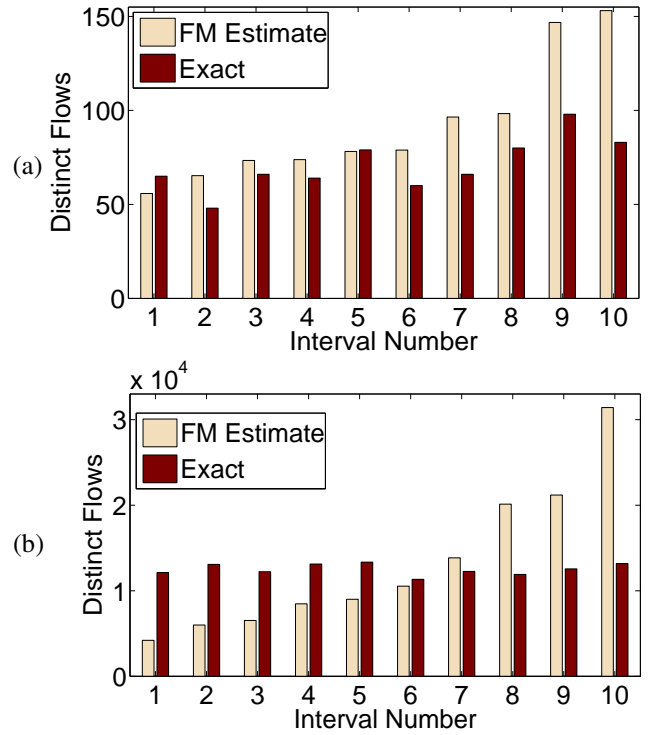


Fig. 3. Comparison of distinct flows (exact) against the values estimated by the FM sketch. (a) FM sketch of ten intervals of normal traffic, (b) FM sketch of ten intervals of malicious traffic.

the vector of length 100 where the  $i^{th}$  entry is 1 if change is detected at  $i$  interval. Similarly, we have a vector  $V_{exact}$  corresponding to the experiment with exact flow counts. The error is defined as  $\frac{\text{edit distance}(V_{fm}, V_{exact})}{\text{No of 1s in } V_{exact}}$ . We observe that the change detection by the FM sketch is almost keeping track with the exact value. In Fig. 4(b) we compare the performance of two distance function by applying the change detection Algorithm 1 to exact flow counts. To compute the error, we use a vector corresponding to each distance function and use the same formula as above. In Fig. 5, we plot the accuracy of the two different distance functions on a real trace in which we inject large values (to create change) at random intervals. Thus, we are able to verify with certainty the accuracy of our algorithm (in the sense that **we do know** where anomalies occur). The size of each injection is uniformly randomly drawn from  $[10a, 20a]$  where  $a = \max a_i$ ,  $a_i$  is the count in interval  $i$ , and we inject at ten random places of 100 intervals. We see that the distance function that uses the difference of the sums of the windows leads to less accurate results. For three values of threshold, the distance function from Algorithm 2 does some misclassification. This may be due to the fact that, in Algorithm 2, we use a probabilistic distance measure. Whereas in sum-diff distance function, we are using deterministic sums of two windows (which might overlook change in distribution).

**Implementation Issues:** For detecting changes in real-time, sketches might need to be in SRAM for fast memory access. Therefore, per packet memory access could be an issue. For FM sketch, memory reference per packet is 1 through a hash function whereas for Bloom filter, memory references is  $k$  per packet during update. Similarly for Count-Min sketch with

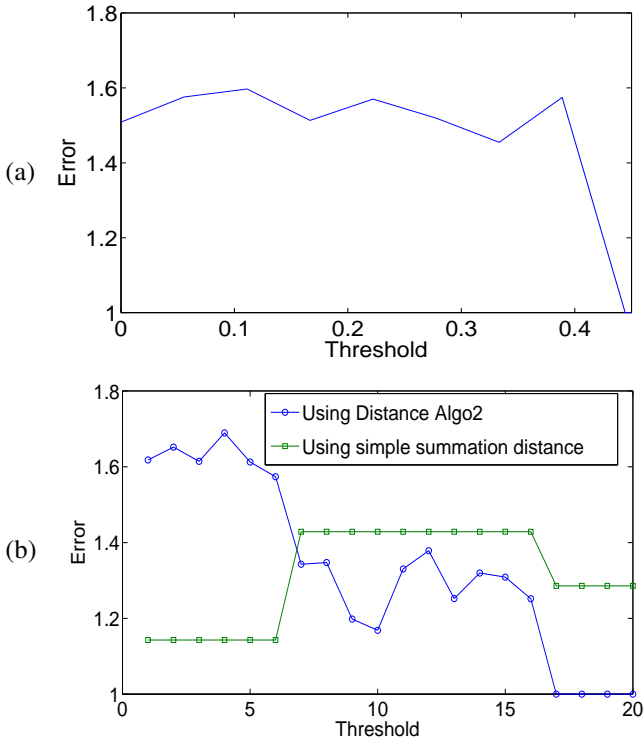


Fig. 4. (a) Change detection accuracy by the FM sketch using distance function (modified from [13]). We choose threshold values for which Error is non-zero. (b) Comparison of two different distance functions on exact flow counts. Through experiments, we select a range of threshold values for which Error is non-zero. The threshold values are different for different distance functions, but normalized to show them on the same scale.

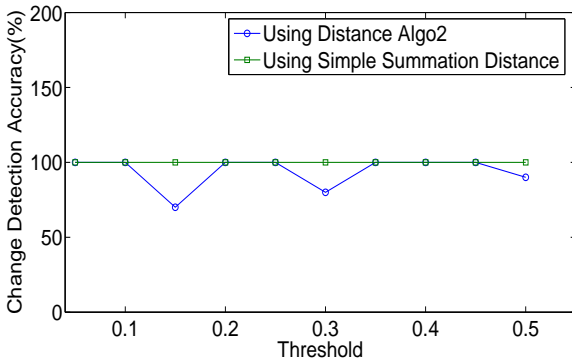


Fig. 5. Accuracy of two distance functions on real traces in which synthetic changes were introduced. The x-axis represents values such as threshold is  $\text{mean}(X) + \alpha \text{std}(X)$  where  $X$  is the flow counts.

depth  $k$ , it is  $k = \lceil \ln \frac{1}{\delta} \rceil$  where  $\delta$  is an input parameter for Count-Min sketch. If one cares about false positive rate in Bloom filter, it is minimized for  $k \approx 0.7 \frac{m}{n}$  where  $m$  is the number bits in the array and  $n$ , the number of inserted elements. There is a trade-off between  $k$  and  $m/n$  given by the false-positive rate as  $(1 - (1 - 1/m)^{kn})^k$ .

## VI. CONCLUSION

Intrusion or attack detection requires knowledge of attack signatures or some invariant characteristics of Internet flows. In this paper, our attempt has been to detect changes in the flows which characterize scanning activities and propagation

of worm or viruses. It is a non-trivial task to evade attacks that are based on spoofing without changing the network architecture or introducing levels of collaboration. We have argued the case for support in the router architecture to detect common Internet attacks. In the evaluation, we find that counting and multi-counting Bloom filters have high accuracy in estimating flow sizes and changes in flow sizes. We find that the FM sketch also estimates the number of distinct flows quite accurately. We observe that, while finding changes in distribution, the distance function affects the accuracy (a simple  $L_2$  distance leads to inaccuracy).

## REFERENCES

- [1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.
- [2] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2004.
- [3] T. Bu, J. Cao, A. Chen, and P. P. Lee. A fast and compact method for unveiling significant patterns in high speed networks. In *Proc. of IEEE INFOCOM*, Alaska, 2006.
- [4] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at Streaming Speeds. In *Proceedings of ACM SIGMOD 2004*, Paris, France, June 2004.
- [5] G. Cormode and S. Muthukrishnan. Improved Data Stream Summary: The Count-Min Sketch and its Applications. Technical report, 2003.
- [6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proceedings of the SIGCOMM 2004*, Portland, Oregon, August 2004.
- [7] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of the SIGCOMM 2003*, pages 137–148, Karlsruhe, Germany, 2003.
- [8] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base algorithms. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
- [9] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 27th VLDB Conference*, Rome, Italy, 2001.
- [10] D. Guo, H. Chen, J. Wu, and X. Luo. Theory and Network Application of Dynamic Bloom Filters. *IEEE Infocom*, 2006.
- [11] M. Handley and A. Greenhalgh. Steps Towards a DoS-resistant Internet Architecture. In *ACM SIGCOMM FDNA'04 Workshop*, August 2004.
- [12] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of ACM SIGCOMM'02*, Pittsburgh, USA, 2002.
- [13] D. Kifer, S. Ben-David, and J. Gehrke. Detecting changes in data streams. In *Proc. of Int. Conf. on Very Large Data Bases*, Canada, 2004.
- [14] B. Krishnamurthy, S. Sen, and Y. Chen. Sketch-based Change Detection: Methods, Evaluation and Applications. In *Proceedings of ACM Internet Measurement Conference'03*, 2003.
- [15] A. Kumar and J. Xu. Sketch Guided Sampling—Using On-Line Estimates of Flow Size for Adaptive Data Collection. *IEEE Infocom*, 2006.
- [16] A. Kuzmanovic and E. W. Knightly. Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of ACM SIGCOMM'03*, 2003.
- [17] A. Lakhina, M. Crovella, and C. Diot. Diagnosing Network-Wide Traffic Anomalies. In *Proceedings of ACM SIGCOMM'04*, 2004.
- [18] LBNL Traffic traces. <http://www.icir.org/enterprise-tracing/>.
- [19] NLANR MyDoom Traffic traces. <ftp://pma.nlanr.net/traces/long/doom/20040126/COS-1075077419-1.tsh.gz>.
- [20] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reverse Hashing for Sketch-based Change Detection in High-Speed Networks. In *Proceedings of ACM/USENIX Internet Measurement Conference'04*, 2004.
- [21] Y. Zhang, Z. M. Mao, and J. Wang. Low-rate TCP-Targeted DoS Attack Disrupts Internet Routing. In *Proceedings of 14th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.