Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks

Sumit Ganguly, Minos Garofalakis, Rajeev Rastogi, Krishan Sabnani

Bell Labs, Lucent Technologies

{sganguly,minos,rastogi,kks}@bell-labs.com

November 29, 2006

Abstract

Effective mechanisms for detecting and thwarting *Distributed Denial-of-Service (DDoS)* attacks are becoming increasingly important to the success of today's Internet as a viable commercial and business tool. Most earlier work on the DDoS-detection problem has typically focused on either off-line analyses of DDoS-attack measurements or on techniques targeting a small number of potential victim destinations; unfortunately, such mechanisms are not useful for detecting possible DDoS activity *in real time* over large ISP networks, where the number of packet destinations to monitor can easily rise to several millions. In this paper, we propose novel data-streaming algorithms for the robust, real-time detection of DDoS activity in large ISP networks. The key element of our solution is a new, hash-based synopsis data structure for network-data streams that allows us to efficiently *track*, in guaranteed small space and time, destination IP addresses in the underlying network that are "large" with respect to the number of *distinct* source IP addresses that have established potentially-malicious (e.g., "half-open") connections to them. Our work is the first to address the problem of efficiently tracking the top distinct-source frequencies over a general stream of *updates* (insertions and deletions) to the set of underlying network flows, thus enabling us to effectively distinguish between DDoS activity and flash crowds. Preliminary experimental results verify the effectiveness of our approach.

Keywords: Data-streaming algorithms, top-k frequencies, denial-of-Service attacks, IP networks, real-time systems.

Technical area: Data management.

Corresponding author's email: rastogi@bell-labs.com.

1 Introduction

Distributed Denial-of-Service (DDoS) attacks are sources of mass disruption in today's Internet. Unlike typical security threats aiming to penetrate security perimeters to steal information, DDoS attacks paralyze Internet systems by swamping network servers, links, and devices (e.g., routers, firewalls) with bogus traffic. A DDoS attack typically directs hundreds or even thousands of compromised "zombie" hosts against a single victim. These zombie hosts are unwittingly recruited from amongst the millions of unprotected computers connected to the Internet and implanted with malicious "sleeper" code awaiting for the command to launch a massive DDoS attack. With a large enough legion of zombie hosts, the volume of such an attack can be astounding.

Packet-flooding attacks that rely on directing large numbers of packets to a specific victim destination are probably one of the most common forms of DDoS attacks. A packet flood can comprise either seemingly legitimate TCP, UDP, or ICMP packets in volumes large enough to overwhelm network devices and servers, or deliberately incomplete packets designed to rapidly consume all available computational resources at the server. Most such attacks also make use of *spoofed source IP addresses* in the packets; that is, they forge the IP address that supposedly generated the request (e.g., using a randomly-chosen address) in order to elude source identification. As a more concrete DDoS scenario, consider the case of a *TCP-SYN-flooding attack* [3]. During such attacks, the attacking host(s) send a flood of (seemingly legitimate) SYN messages with spoofed source IP addresses to establish a TCP connection with the victim server system. The victim responds with a SYN-ACK message but, since the source IP address in the original packet was spoofed, the final step in establishing the TCP connection (i.e., an ACK message from the client) is never sent to the victim. This sequence essentially creates a *"half-open"* TCP connection at the victim server. Now, the server needs to maintain a finite-size, in-memory data structure describing all pending connections. By intentionally creating a persistent, large number of half-open connections with a continual flood of spoofed SYN packets, this data structure can be made to overflow. This, in turn, causes the server to reject any incoming legitimate connection requests for the duration of the attack and, in some cases, can force the server to exhaust its memory, crash, or be rendered otherwise inoperative [3].

Prior Work. The impact of successful DDoS attacks is severe and widespread. Compromised server performance can result in Service-Level-Agreement (SLA) violations, frustrated customers, and cumulative losses that can easily mount to millions or even billions of dollars for large corporations [1]. As a result, the problem of effectively detecting and mitigating DDoS activity has received a lot of interest from several recent research as well as commercial efforts. The bulk of this earlier work has typically concentrated on either *off-line analysis* trying to infer some interesting patterns of DDoS activity after the fact, or simple filtering and statistical techniques that try to identify traffic anomalies for a small number of suspected victim destinations or suspicious sources [4, 26, 20, 27, 21, 29]. Clearly, off-line techniques are

not very useful for on-line, real-time DDoS detection. Similarly, techniques that rely on monitoring traffic patterns for specific source/destination addresses are impossible to scale when it comes to detecting DDoS activity over the network of a large Internet Service Provider (ISP) – even for a single router in the ISP's backbone, the set of potential addresses to monitor can easily rise to millions, and maintaining per-address state quickly becomes infeasible. Scalability issues also plague the approaches in [31, 11, 34, 30] because they require a certain amount of memory to be allocated for each distinct source-destination pair [31], each source [11], or each flow [34, 30].

Estan and Varghese [10] have suggested DoS-detection algorithms that operate with small-space and small-time guarantees in a *data-streaming* fashion [16]; that is, they operate by looking at the stream of packets *only once* (in the fixed order of arrival) and guarantee a small memory footprint (much smaller than the domain of distinct addresses) and low processing time per packet. In a nutshell, their algorithms employ ideas based on sampling [17, 25] and hash-based filtering [6] to identify (with high probability) *large flows* (i.e., flows occupying more than a certain fraction of the monitored network link). Still, it is unclear whether identifying large flows is a robust indicator of DDoS activity. As an example, in the TCP-SYN-flooding scenario described earlier in this section, it is easy to see that none of the malicious, half-open TCP flows will be large since no data packets are ever exchanged. An additional concern with such "large-flow" techniques is that, by tracking only the volume of flow traffic, they make it impossible to distinguish between DDoS attacks and *flash crowds* representing an unexpected surge of legitimate requests to a server due to an important event (e.g., the 9/11 terrorist attack in the U.S.). The recent proposal of Akella et al. [4] also suffers from similar limitations and may fail to detect DDoS attacks, since it relies on maintaining profiles (e.g., total number of bytes, number of distinct source prefixes sending traffic) only for so-called *popular* destinations, whose traffic exceeds a certain threshold.

Wang et al. [36] have proposed a SYN-flood detection mechanism that relies on monitoring the difference between the aggregate number of TCP SYN and FIN/RST packets at a leaf router, and employing statistical (i.e., Sequential Change Point Detection) techniques to detect abrupt changes (potentially indicating a SYN-flooding attack). Their algorithms must be run on individual first- or last-mile routers, and cannot be used to detect signs of *distributed* attacks (or, identify potential victims) in large ISP networks; furthermore, their statistical change-detection techniques are, in a sense, complementary to the results and tools presented in this paper. Krishnamurthy et al. [23] explore sketch-based methods to detect significant changes in massive data streams with a large number of flows.

For scalable detection of attacks like TCP-SYN-flooding or various port scans in high-speed networks, [22, 15] maintain multiple parallel hash tables with bucket counters to keep track of sources/destinations that hash into each bucket. The idea is that if a destination (or source) hashes into buckets with large counters in all the hash tables, then there is a good chance that the destination is being attacked. [24] discusses new types of hard-to-detect DoS attacks

based on low-rate traffic patterns that exploit TCP's retransmission time-out mechanism. Recently, to limit DoS attack traffic, there have been proposals based on a capability-based architecture [35], and encouraging "good" clients to send more traffic, thereby crowding out the "bad" ones [33]. However, none of the above papers considers new streaming algorithms (with provable guarantees) for detecting DDoS attack scenarios in real time.

The recent work of Venkataraman et al. [32] is most closely related to our work. It proposes streaming algorithms for identifying *k-superspreaders*, sources that connect to more than k distinct destinations for a given threshold k. We provide a more detailed comparison after describing our approach below.

Our Contributions. In this paper, we consider the problem of robust, real-time detection of DDoS activity over large ISP networks. The key element of our approach is a novel data-streaming algorithm for efficiently tracking, in guaranteed small space and time, destination IP addresses that are "large" with respect to the *number of distinct source IP addresses that access them*. We believe that such a *distinct-source frequency* metric for a destination IP address provides a very robust indicator of potential DDoS activity targeted at the destination – this is clearly the case for the TCP-SYN-flooding attack scenario described earlier or any other DDoS attack employing source IP-address spoofing. We propose new data streaming algorithms and synopsis data structures for effectively tracking the *top-k* destinations with respect to the distinct-source frequency metric over a large ISP network. Our techniques rely on simple, hash-based sketch synopses of the stream of IP packets with small memory footprint and a guaranteed small (i.e., logarithmic) number of simple maintenance operations per packet.

Furthermore, our tracking algorithms can readily handle *deletions* in the data stream, which provides us with a very effective way of distinguishing between malicious flows (which should be accounted for in our frequency counting) and legitimate flows (which should be ignored). The idea here is that flows that have been established as legitimate, for example, by client sources returning an ACK packet to the server in the SYN-flooding scenario, should be removed from our synopses, since our primary goal for SYN-flood detection is to estimate the number of distinct source IP addresses with *half-open* TCP connections. Thus, by effectively handling deletions as well as insertions in the input data stream, our techniques provide an effective mechanism for distinguishing between flash crowds and SYN-flooding attacks. To the best of our knowledge, our work is the first to address the problem of efficiently tracking the top-k distinct frequencies over a stream of updates (i.e., insertions and deletions), and explore its application in the context of DDoS detection[1]. More concretely, our contributions can be summarized as follows.

• **Novel, Small Update Time Stream Synopsis for Estimating Top Distinct Frequencies.** We propose a novel hash-based synopsis data structure, the *Distinct-Count Sketch*, for estimating the top-k distinct-source frequencies. A distinct-

[1]Our top-k distinct frequencies tracking algorithms can also be used to identify hosts that contact many distinct destinations during port scans (mostly for worm propagation).

count sketch imposes a small space overhead, and can be efficiently maintained by performing a guaranteed small (i.e., logarithmic) number of simple hash operations per element in the stream. Furthermore, unlike earlier distinct-count estimation techniques (e.g., the *distinct samples* of Gibbons et al. [18, 19], or the *cascaded summaries* of Cormode and Muthukrishnan [8] for multigraph streams), our distinct-count sketch synopsis can readily handle deletions in the stream, making it ideal for real-time detection of DDoS-attack scenarios like TCP-SYN-flooding. Also, our top-k problem is different from the *k-superspreaders* problem [32] whose objective is to identify sources that connect to more than k distinct destinations for a given threshold k. Our problem, on the other hand, seeks to find the top-k destinations connected to the most distinct sources. Thus, in our setting, users are not required to specify threshold values on the number of distinct connections for a source/destination which can be difficult to determine in practice. Further, our distinct-count sketch synopsis is novel, and is designed to be able to quickly estimate the top distinct frequencies in the presence of stream deletions.

• **Continuous-Tracking Algorithm for Top-k Distinct-Frequency Destinations.** We develop the first known streaming algorithms for *continuously tracking* the top-k destinations with the largest distinct frequencies. As with our basic distinct-count sketches, our *Tracking Distinct-Count Sketch* synopses incur a small (logarithmic) number of steps to process each streaming update. Furthermore, at any given point in the stream, our tracking algorithms are able to produce, in *guaranteed logarithmic time*, an approximate set of top-k destinations (and, corresponding distinct frequencies) that is provably close (with high probability) to the actual top-k set. [2] Consequently, our tracking algorithms can be readily deployed to monitor large ISP networks transiting large volumes of IP packet data.

• **Experimental Results Validating our Approach.** We present the results of an experimental study that demonstrate the effectiveness of our tracking algorithms for estimating the top-k frequencies. Specifically, our results indicate that even with a very small distinct-count sketch (whose size is only a small fraction of the complete information needed to exactly capture the distinct-source frequencies), we are able to estimate the large frequencies with low relative errors. Furthermore, our tracking algorithms are extremely fast, incurring only a few tens of microseconds to process each stream update.

Due to space constraints, some of our theoretical results are presented here without proof. All the details can be found in the full version of this paper [14].

[2] This is in sharp contrast with earlier work on hash-based sketches for update streams [13] which cannot guarantee small update *or* query-tracking times.

2 System Model and Problem Formulation

In this section, we describe our general stream-processing architecture for detecting DDoS activity in a large ISP network and formally define the DDoS-detection problem addressed in this paper.

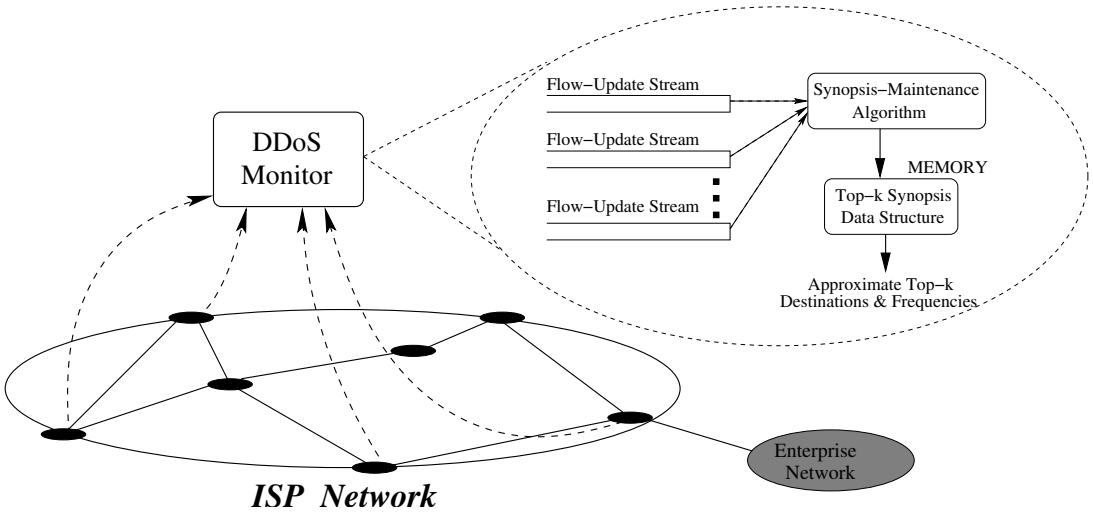

Figure 1: Update-Stream Processing Architecture.

Stream-Processing Model. The key elements of our stream-processing architecture for DDoS detection are depicted in Figure 1. In a nutshell, our DDOS MONITOR box monitors flow information in the underlying network by observing a (collection of) continuous streams of *flow updates* from various elements in the underlying ISP network. Each such flow update can be abstracted as a triple of the form $(source, dest, \pm 1)$, where: (1) $(source, dest)$ is a source-destination IP address pair, interpreted as an indicator of a flow connection from *source* to *dest*, and (2) ± 1 denotes the net change in the frequency of potentially-malicious $(source, dest)$ flows. For example, in the SYN-flooding scenario described in Section 1, the original SYN packet from *source* to *dest* appears with a "+1" in the flow-update stream (i.e., insertion), whereas the corresponding ACK packet establishing the legitimacy of the TCP connection would appear as a "-1" flow-update triple (i.e., deletion). Note that such input flow-update streams to our DDOS MONITOR can be generated using a variety of network-monitoring tools, e.g., by deploying Cisco's NetFlow tool [2] or AT&T's recently-proposed GigaScope probe [9] to monitor egress-flow traffic (and corresponding TCP flags) for routers at the edge of the ISP network. Finally, even though our DDOS MONITOR tool is depicted outside the network in Figure 1, our proposed techniques are fairly lightweight in terms of both memory footprints and processing times, making them also suitable for deployment inside the network; for example, to monitor the packet streams seen at an ISP-backbone router for signs of DDoS activity.

Without loss of generality, we assume that each (source or destination) IP address in the input flow-update stream(s)

Symbol	Semantics
(u, v)	Generic source-destination IP address pair
m (m^2)	Size of integer domain for IP addresses (resp., IP address pairs)
n	Total number of elements in the stream(s)
f_v	(Distinct-source) frequency of destination v
$v_1, \ldots, v_k$	Current top-k destinations wrt frequency
$U = \sum_i f_{v_i}$	Total number of distinct source-destination address pairs in the streams

Table 1: Notation.

takes values in the integer domain $[m] = \{0, \ldots, m-1\}$, so each $(\textit{source}, \textit{dest})$ address pair takes values in the integers $[m^2]$ (e.g., by concatenating the two addresses in the pair). We also let n denote an upper bound on the total size (i.e., number of update tuples) in the input streams. (Table 1 summarizes some of the key notational conventions used in the paper.) Given the volume of flow-update streams in large ISP networks (e.g., AT&T's IP backbone alone generates 500 GBytes of NetFlow data per day [9]) and the need for real-time reaction to DDoS attacks, our DDoS MONITOR tool needs to operate in a *streaming* fashion [16] – that is, it is only allowed to see the flow-update tuples *only once*, in the fixed order of arrival from the network, and explicit backtracking (i.e., access to past tuples) over the update stream is impossible. Abstractly, our DDoS MONITOR aims to identify (in real-time) destination IP addresses that are potentially under a DDoS attack based on the observed flow-update streams. Given the scale of large ISP networks, an important design goal for our DDoS MONITOR is that it *cannot maintain state for each source-destination in the network*; clearly, maintaining a potential 2^{64} counters would make our solution far too heavyweight in terms of both space and update time. Instead, we allow our DDoS MONITOR only a certain amount of memory (typically, significantly smaller than m and n), which is used to maintain a concise *synopsis* of the input flow-update stream(s) (Figure 1). The key constraints imposed on our stream synopsis are that: (1) it is significantly smaller than the number of distinct addresses in the network (e.g., its size is logarithmic or poly-logarithmic in m); (2) it can be easily maintained in small (i.e., logarithmic or poly-logarithmic) time per update tuple during a single pass over the update stream in the (fixed) order of arrival; and, (3) it continuously *tracks* the quantities of interest, i.e., it can readily provide an answer to our DDoS query at any point during the stream.

Problem Formulation. The key goal of our DDoS MONITOR tool is to track (in small space/time) over the flow-update stream(s) the top-k destination IP addresses in the network with respect to the *number of distinct source IP addresses* that have established potentially-malicious (e.g., half-open) TCP connections to them. More formally, in our flow-update stream setting described above, given a destination IP address v in the underlying network, we define the *distinct-source frequency* of v (denoted by f_v) as the number of distinct source addresses u for which the net number of occurrences

of the (u, v) source-destination pair in the input update streams is positive; that is, $f_v = |\{u : \text{OCCUR}(u, v, +1) > \text{OCCUR}(u, v, -1)\}|$, where $\text{OCCUR}(t)$ denotes the number of times the update tuple t is seen in the input stream(s). As already discussed in Section 1, such a distinct-source frequency metric can provide a fairly robust indicator of DDoS activity (e.g., in a SYN-flooding scenario), and the "largest" destinations in the underlying network are clearly the most likely victims of such activity. Thus, by continuously tracking the top-k distinct-source frequency destinations over the stream of flow updates, our DDoS MONITOR can readily identify (in real time) signs of potential DDoS activity in the network (e.g., by comparing against "baseline" profiles of network activity created over longer periods of time). [3] To simplify the discussion, we often refer to "distinct-source frequency" simply as "frequency" in the remainder of this paper.

Without loss of generality, assume that the list of destination addresses in the underlying network is $v_1, v_2, \ldots$ with destinations sorted with respect to the frequency metric, i.e., $f_{v_1} \geq f_{v_2} \geq \ldots$. Given a target number k of most frequent destinations in the update stream, our goal is to continuously track the top-k destinations $v_1, \ldots, v_k$ and their corresponding frequencies $f_{v_1}, \ldots, f_{v_k}$. Note, of course, that the actual list of top-k destination addresses as well as their frequencies can vary throughout the duration of the input stream(s); more generally, all the symbols defined in Table 1 and used in our analysis are defined, at any given point in time, with respect to the stream of updates seen thus far. Strong lower-bound results exist even for simple variants of our tracking problem. For example, Alon et al. [5] show that any algorithm for tracking the most popular (i.e., most-frequently occurring) address in an (insert-only) stream to within constant relative error with constant proability requires $\Omega(m)$ space[4]. Thus, in this paper, we focus on the following (approximate) variant of the top-k destination-tracking problem for which our techniques can achieve significant space savings, and guaranteed small update and query times.

TRACKAPPROXTOPK($\mathcal{S}$, k, ϵ, δ)

- **Input:** Flow-update stream(s) $\mathcal{S}$, number of desired top-frequency elements k, desired relative error ϵ and probabilistic confidence δ in the frequency estimates.
- **Output:** Continuously track a list L of k destination IP addresses from $\mathcal{S}$ such that, at any point in the stream, we are guaranteed that with probability of at least $1 - \delta$:
  1. Any destination address $v \in L$ has frequency $f_v \geq (1 - \epsilon) f_{v_k}$; and,
  2. For any destination $v \in L$, the relative error in our frequency estimate $\hat{f}_v$ is at most ϵ, i.e., $\Pr\left[|\hat{f}_v - f_v| \leq \epsilon f_v\right] \geq 1 - \delta$.

In other words, our approximate top-k tracking algorithm guarantees that (with high probability) only destinations with frequencies very close to f_{v_k} are actually output (Clause 1); furthermore, all the output destination frequency

[3] Even though our discussion here focuses on top-k tracking, our techniques and results also easily extend to the problem of tracking all destinations v with $f_v \geq \tau$, for some fixed threshold τ.

[4] The asymptotic notation $f(n) = \Omega(g(n))$ is equivalent to $g(n) = O(f(n))$. Similarly, the notation $f(n) = \Theta(g(n))$ means that functions $f(n)$ and $g(n)$ are asymptotically equal (to within constant factors); in other words, $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$ [7].

estimates are within a relative error of ϵ with probability $\geq 1 - \delta$ (i.e., (ϵ, δ)-*estimates*) of their true values (Clause 2).

3 The Distinct-Count Sketch Synopsis for Flow-Update Streams

In this section, we present our basic, novel stream-synopsis data structure (termed *Distinct-Count Sketch*) and the algorithm for maintaining a distinct-count sketch over continuous streams of flow updates.

The Distinct-Count Sketch Data Structure. Our distinct-count sketch synopsis is a non-trivial generalization of the basic bit-vector hash structure proposed by Flajolet and Martin for the simple problem of distinct-value estimation [12]. A distinct-count sketch employs a randomly-chosen hash function h that (as in [12]) maps the domain of source-destination address-pair values $[m^2]$ onto a logarithmic range $\{0, \ldots, \Theta(\log m)\}$ of buckets with exponentially-decreasing probabilities. That is, $h : [m^2] \rightarrow \{0, \ldots, \Theta(\log m)\}$, with $\Pr[h(x) = l] = 1/2^{l+1}$ for any $x \in [m^2]$ – thus, we expect $1/2$ of the distinct values in $[m^2]$ mapping to bucket 0, $1/4$ mapping to bucket 1, and so on.[5] For each partition of the $[m^2]$ domain corresponding to such a (first-level) hash bucket, we maintain an array of r (second-level) hash tables; for each $i = 1, \ldots, r$, the i^{th} second-level hash table comprises s buckets and employs a randomizing hash function g_i (the same across all first-level buckets) that maps $[m^2]$ uniformly onto $[s]$ (i.e., $g_i : [m^2] \rightarrow [s]$) for mapping elements of the first-level partition onto second-level hash buckets. Here, the g_i hash functions are assumed to be mutually independent (e.g., defined using independently-chosen random seeds); also, note that the values of the parameters r and s defining the number and size, respectively, of the inner hash tables are fixed later (based on our analysis). Finally, each second-level hash bucket in our distinct-count sketch maintains a small (logarithmic-size) *count signature* for the (multi)set of source-destination pairs that are mapped onto this bucket.

The second-level hash buckets are essentially needed to ensure that our synopsis structure is resilient to deletes. In addition to keeping track of the number of source-destination pairs that map to each bucket, the count signatures in the buckets also help to (1) detect collisions (that is, instances when multiple elements hash to the same bucket), and (2) identify the source-destination pair mapping to a bucket in case there no collision. The count signature maintained for each second-level hash bucket is basically an array of $\log m^2 + 1 = 2 \log m + 1$ counters (each of size $\Theta(\log n)$), comprising two parts: (a) one *total element count*, which tracks the net total number of source-destination pairs that map onto the bucket; and, (b) $2 \log m$ *bit-location counts*, which track, for each $j = 1, \ldots, 2 \log m$, the net total number of

[5] Such a hash function h can be easily implemented using a function f that randomizes values of $[m^2]$ uniformly over $[m^{2k}]$ (where k is a small constant, e.g., $k = 2$, used to guarantee that the mapping over $[m^{2k}]$ is injective with high probability), combined with the LSB operator that returns the *least-significant 1 bit* in a binary string. Simply defining $h(x) = \text{LSB}(f(x))$ gives a hash function with the desired properties [12].

source-destination pairs $(u, v) \in [m^2]$ with $\text{BIT}_j(u, v) = 1$ that map onto the bucket (where, $\text{BIT}_j(u, v)$ denotes the value of the j^{th} bit in the binary representation of the (u, v) source-destination pair). Conceptually, a distinct-count sketch synopsis can be seen as a four-dimensional array $\mathcal{X}$ of size $\Theta(\log m) \times r \times s \times (2 \log m + 1) = r \times s \times \Theta(\log^2 m)$, where each entry $\mathcal{X}[i, j, k, l]$ is a source-destination pair counter of size $\Theta(\log n)$ corresponding to the l^{th} count-signature location of the k^{th} second-level hash bucket in hash table number j for the i^{th} first-level hash bucket. (Note that the total space requirement of a distinct-count sketch is $\Theta(rs \log^2 m \log n)$.) By convention, we assume that, for a given second-level hash bucket (i, j, k), $\mathcal{X}[i, j, k, 0]$ is always the total source-destination pair count, whereas the bit-location counts are located at $\mathcal{X}[i, j, k, 1]$, ..., $\mathcal{X}[i, j, k, 2 \log m]$. The structure of our distinct-count sketch synopsis for flow-update streams is pictorially depicted in Figure 2.

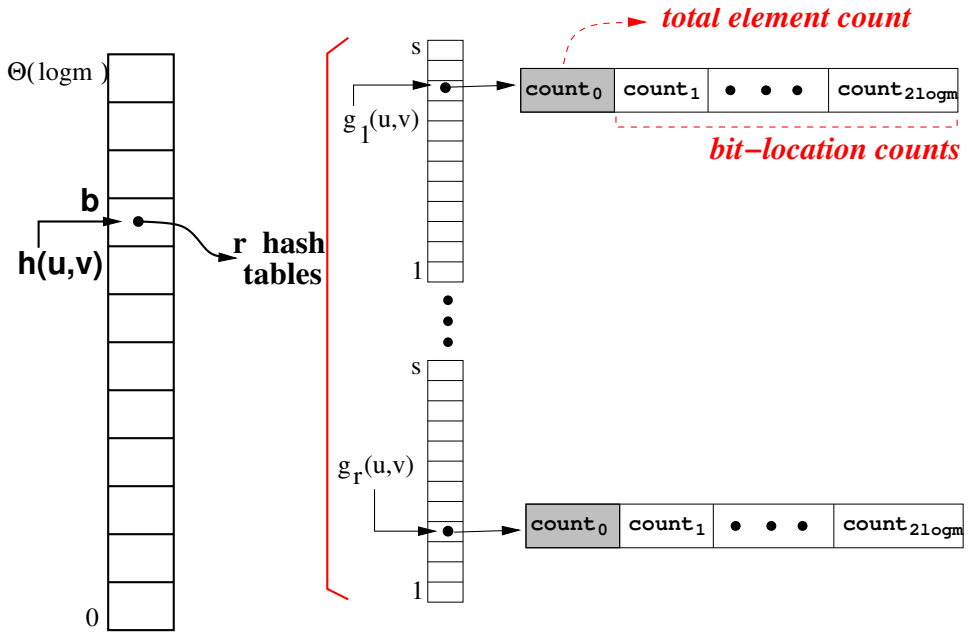

Figure 2: Our Distinct-Count Sketch Update-Stream Synopsis.

Maintenance. The algorithm for maintaining a distinct-count sketch synopsis $\mathcal{X}$ over (one or more) streams of flow updates is fairly straightforward. The sketch structure is first initialized to all zeros and, for each incoming update $(u, v, \pm 1)$, the element counters at the appropriate locations of the $\mathcal{X}$ sketch are updated. More specifically, for each second-level hash table $j = 1, \ldots, r$ of the first-level hash bucket identified by $h(u, v)$, we simply set $\mathcal{X}[h(u, v), j, g_j(u, v), 0] := \mathcal{X}[h(u, v), j, g_j(u, v), 0] \pm 1$ to update the total count in the corresponding second-level hash bucket and, for each $l = 1, \ldots, 2 \log m$ *such that* $\text{BIT}_l(u, v) = 1$, we set $\mathcal{X}[h(u, v), j, g_j(u, v), l] := \mathcal{X}[h(u, v), j, g_j(u, v), l] \pm 1$ to update the appropriate bit-location counts in the bucket. Note here that our distinct-count sketch synopsis is essentially impervious

to delete operations; in other words, the sketch obtained at the end of an update stream is *identical* to a sketch that never sees the deleted items in the stream. Also, it is easy to see that the maintenance time per streaming flow update for a distinct-count sketch is only $O(r \log m)$.

4 Top-k Frequency Estimation using a Distinct-Count Sketch

In this section, we present our baseline estimation procedure for approximating the top-k distinct-source frequencies over a set of flow-update streams using a distinct-count sketch synopsis, along with a detailed analysis of its space/accuracy guarantees. More specifically, assume a distinct-count sketch $\mathcal{X}$ maintained over the input flow-update stream(s) (Figure 1), as described in Section 3. Our baseline estimator for the top-k distinct-source frequencies (and corresponding destination addresses), termed BaseTopk, is depicted in Figure 3. In a nutshell, the main idea in our estimator is to employ the distinct-count sketch synopsis to build an (appropriately-sized) *distinct sample* of the observed active source-destination pairs. Unlike traditional random sampling, a distinct sample is essentially a uniform random sample collected over the *distinct values* of the underlying domain, thus enabling accurate estimation for quantities with *set (i.e., frequency-independent) semantics* (like our distinct-source frequencies); see, for example, [18, 19].[6] Once such a distinct sample (dSample) is available, our top-k estimation process is fairly straightforward: the destinations with the k highest occurrence frequencies in dSample are identified, and are returned with their frequencies appropriately scaled by the distinct-sampling rate (Steps 8–9).

Building the distinct sample (dSample) of source-destination pairs from $\mathcal{X}$ proceeds iteratively, starting from the topmost first-level hash bucket of $\mathcal{X}$ and dropping to lower buckets until dSample reaches a desired size (Steps 1–6). More specifically, our target size for dSample is $\Theta(s)$ (Step 3), where, of course, s denotes the number of second-level hash buckets in $\mathcal{X}$ (Section 3). (Our analysis shows that this target sample size is always reached with high probability.) At each bucket level b, our estimator uses the GetdSample$(\mathcal{X}, b)$ procedure to extract a new set of sample points from the current bucket and add these points to the running dSample (Step 4).

The pseudo-code for our GetdSample procedure is shown in Figure 4. The key idea here is to employ the second-level hash tables and count signatures in our distinct-count sketch synopsis in order to identify (the binary signatures of) specific source-destination address pairs that have mapped onto the specified first-level bucket b; these pairs comprise our output distinct sample from this bucket. To identify such pairs, our GetdSample procedure essentially iterates over all second-level hash buckets (j, k) for bucket b, using the count-signature information to discover second-level buckets

[6] In a sense, our distinct-count sketch synopsis can be seen as a distinct-sampling technique that, unlike the earlier methods of Gibbons et al. [18, 19], is completely *delete-resistant*.

Procedure BaseTopk($\mathcal{X}, \epsilon$)
Input: Distinct-Count Sketch synopsis $\mathcal{X}$ over input stream(s), relative accuracy parameter ϵ.
Output: Approximate top-k destinations and corresponding distinct-source frequencies.
begin
1. b := index of topmost first-level hash bucket of $\mathcal{X}$ // $b = \Theta(\log m)$
2. dSample := ϕ
3. **while** ($b \geq 0$ **and** |dSample| $< (1+\epsilon)s/16$) **do** {
4. dSample := dSample$\cup$ GetdSample($\mathcal{X}, b$)
5. $b := b - 1$
6. }
7. // dSample = *distinct sample* of source-dest (u, v) pairs
8. let $v_1, \ldots, v_k$ be the k destinations with the highest occurrence frequencies (say, $f^s_{v_1}, \ldots, f^s_{v_k}$) in dSample
9. **return** $\{< v_i\ ,\ 2^b \cdot f^s_{v_i} >:\ 1 \leq i \leq k\}$
end

Figure 3: Baseline Top-k Estimator using a Distinct-Count Sketch.

that are *singletons* (i.e., contain only a single distinct source-destination pair) (Steps 2–6). The key here is that, based on our count-signature structure, a second-level bucket is a singleton iff *for each bit-location* $l = 1, \ldots, 2\log m$, the corresponding bit-location count is either zero (i.e., $\mathcal{X}[b, j, k, l] = 0$, implying only pairs with a 0 bit in that location) or equal to the total element count (i.e., $\mathcal{X}[b, j, k, l] = \mathcal{X}[b, j, k, 0]$, implying only pairs with a 1 bit in that location). Furthermore, this allows us to trivially identify the (binary signature of the) unique address pair in a singleton bucket. Procedure ReturnSingleton (also depicted in Figure 4) basically implements this logic to identify and return an address pair from a singleton second-level bucket, and return **null** in the case of an empty bucket or if there is a *collision* (i.e., ≥ 2 address pairs map onto the bucket).

Analysis. Deriving the analytical space/accuracy guarantees for our BaseTopk estimation procedure relies on a couple of interesting observations regarding our distinct-count sketch synopsis structure $\mathcal{X}$. (Throughout our analysis, we use r and s to denote the number and size, respectively, of the inner hash tables for each first-level bucket in $\mathcal{X}$.) First, recall (from Section 3) that the expected number of source-destination address pairs in a first-level hash bucket drops exponentially with the bucket index (i.e., $\Pr[h(x) = l] = 1/2^{l+1}$ for any $x \in [m^2]$). Thus, we typically expect to find only few distinct address pairs in the higher first-level bucket indexes which, of course, also implies that pairs in such buckets are unlikely to collide in the corresponding second-level hash tables. Second, even for first-level buckets in our sketch where the number of distinct mapped address pairs is $\Theta(s)$ (i.e., our target distinct-sample size), the second-level randomization of these pairs across several, independently-built hash tables each with s buckets, implies that each such pair appears as a singleton in some second-level bucket (and, thus, is recovered in our distinct sample) with high probability. Finally, the above two observations coupled with the fact that our distinct-sampling procedure proceeds top-

Procedure GetdSample$(\mathcal{X}, b)$
Input: Distinct-Count Sketch $\mathcal{X}$, first-level bucket index b.
Output: Distinct sample ds of source-destination pairs from bucket b.
begin
1. ds := ϕ
2. **for** j := 1 **to** r **do**
3. **for** k := 1 **to** s **do** {
4. (u, v) := ReturnSingleton$(\mathcal{X}, b, j, k)$
5. **if** ($(u, v) \neq$ **null**) **then** ds := ds $\cup \{(u, v)\}$
6. }
7. **return** ds
end

Procedure ReturnSingleton$(\mathcal{X}, b, j, k)$
Input: Distinct-Count Sketch $\mathcal{X}$, index triple (b, j, k) specifying a second-level hash bucket $\mathcal{X}[b, j, k, *]$ in $\mathcal{X}$.
Output: (Binary signature of) the single source-destination pair in $\mathcal{X}[b, j, k, *]$ if that bucket is a *singleton*; **null**, otherwise.
begin
1. **if** ($\mathcal{X}[b, j, k, 0] = 0$) **return null** // bucket is empty
2. **for** l := 1 **to** $2 \log m$ **do** {
3. **if** ($\mathcal{X}[b, j, k, l] = \mathcal{X}[b, j, k, 0]$) **then** set $\text{BIT}_l(u, v) := 1$
4. **else if** ($\mathcal{X}[b, j, k, l] = 0$) **then** set $\text{BIT}_l(u, v) := 0$
5. **else return null** // *collision*: ≥ 2 pairs in the bucket
6. }
7. **return** (u, v)
end

Figure 4: Computing a Distinct Sample from a First-Level Bucket.

down across the first-level buckets of $\mathcal{X}$ imply that we reach our $\Theta(s)$ target sample size with high probability; then, a simple application of Chernoff bounds [28] (with an appropriate setting for s) gives us the space/accuracy guarantees for our BaseTopk estimator. In the remainder of this section, we formalize and prove the above informal claims. To simplify the exposition, we often abstract away the detailed constants from our analysis using Θ-notation; the exact constants can be easily worked out and are quite small for all practical purposes.

Let $U = \sum_i f_{v_i}$ denote the total number of distinct source-destination address pairs with positive net frequency in our input flow-update streams $\mathcal{S}$. Furthermore, given a first-level bucket index b, let u_b denote the number of such distinct address pairs mapping onto first-level buckets with indexes $\geq b$, and let ds_b be the distinct sample collected by our estimator from first-level buckets with indexes $\geq b$. Note that u_b is a random variable with expectation

$$\mathbf{E}[u_b] = U \cdot \left(\frac{1}{2^{b+1}} + \frac{1}{2^{b+2}} + \cdots \right) = \frac{U}{2^b},$$

since the probability of each distinct pair mapping to the l^{th} first-level bucket in $\mathcal{X}$ is $1/2^{l+1}$ (Section 3). Then, by Chernoff bounds, we have that $\Pr\left[|u_b - \frac{U}{2^b}| > \epsilon \frac{U}{2^b}\right] \leq 2\exp(-\frac{\epsilon^2 U}{2 \cdot 2^b})$, or, equivalently,

$$\Pr\left[|u_b - \frac{U}{2^b}| < \epsilon\frac{U}{2^b}\right] \geq 1-\delta, \text{ if } \frac{U}{2^b} \geq \Theta(\frac{\log(1/\delta)}{\epsilon^2}). \quad (1)$$

Our first lemma formalizes our earlier claim, showing that, if the number of elements mapped to bucket indexes $\geq b$ is at most $s/2$, then our distinct-sampling procedure recovers all these elements with high probability as long as there are $r = \Theta(\log(n/\delta))$ independent inner hash tables. This is because each element appears as a singleton in a second-level hash bucket with constant (1/2) probability, and over $r = \Theta(\log(n/\delta))$ second-level hash tables, this probability grows to $1 - \delta/n$. Thus, every element is included in the sample with high probability.

Lemma 4.1 *Let $r = \Theta(\log(n/\delta))$ and assume that $u_b \leq s/2$. Then, Pr $[|ds_b| = u_b] \geq 1-\delta$.* ∎

Our second lemma now demonstrates that the stopping condition requiring a $\Theta(s)$ distinct sample in our BaseTopk estimator (Step 3) is satisfied with high probability at a first-level bucket index b such that $\frac{U}{2^b} \in [\frac{s}{16}, \frac{s}{4}]$.

Lemma 4.2 *Let $\epsilon < 1/3$, $r = \Theta(\log(n/\delta))$, and $s \geq \frac{16\cdot\log((\log m)/\delta)}{\epsilon^2}$. Also, let b be the highest bucket index such that $u_b \geq \frac{(1+\epsilon)\cdot s}{16}$. Then, with probability at least $1-\delta$, (a) $s/16 \leq \frac{U}{2^b} \leq s/4$, and (b) $|ds_b| = u_b$.* ∎

We are now in a position to demonstrate our main analytical result stating the key accuracy/space guarantees for the frequency estimates returned by our BaseTopk estimator. Given a destination IP address v, let f_v^s denote the occurrence frequency of destination v in the distinct sample collected by our BaseTopk procedure, and let b denote the first-level bucket index at which the distinct-sampling loop of BaseTopk terminates – note that the estimates $\hat{f}_v$ returned by BaseTopk are simply $\hat{f}_v = 2^b \cdot f_v^s$ (Figure 3).

Lemma 4.3 *Let $\epsilon < 1/3$, $r = \Theta(\log n/\delta)$, and $s \geq \frac{16\cdot\log((n+\log m)/\delta)\cdot U}{\epsilon^2\cdot f_{v_k}}$. Then, each frequency estimate $\hat{f}_v$ computed by our* BaseTopk *procedure satisfies: $|\hat{f}_v - f_v| \leq \epsilon\max\{f_v, f_{v_k}\}$, with probability $\geq 1-\Theta(\frac{\delta}{n})$.* ∎

Proof: By Lemma 4.2, we know that (with high probability) the stopping bucket index b satisfies $\frac{U}{2^b} \geq s/16$ and, furthermore, all pairs mapped onto buckets $\geq b$ make it in our distinct sample (i.e., $|\text{ds}_b| = u_b$). Fix a destination v with frequency f_v. Since distinct IP address pairs map onto buckets with index $\geq b$ with probability $1/2^b$, we know that the random variable f_v^s (i.e., the occurrence frequency of destination v in our distinct sample) has expectation

$$\mathbf{E}\left[f_v^s\right] = \frac{f_v}{2^b} \geq \frac{f_v \cdot s}{16 \cdot U} \geq \frac{f_v}{f_{v_k}} \cdot \frac{\log((n+\log m)/\delta)}{\epsilon^2}$$

(using the assumed lower bound on s). Thus, by Chernoff bounds, our frequency estimate $\hat{f}_v = 2^b \cdot f_v^s$ satisfies

$$|\hat{f}_v - f_v| \leq \left(\epsilon\sqrt{\frac{f_{v_k}}{f_v}}\right) \cdot f_v \leq \epsilon \cdot \max\{f_v, f_{v_k}\},$$

with probability $\geq 1 - \frac{\delta}{n}$. This completes the proof. ∎

Finally, Theorem 4.4 follows directly from Lemma 4.3 and concludes our analysis by identifying the space and update/query-time requirements under which our BaseTopk estimator returns robust (ϵ, δ)-approximations for the top-k destinations.

Theorem 4.4 *Let $\epsilon < 1/3$. Procedure* BaseTopk *returns a list of (ϵ, δ) frequency estimates satisfying Clauses (1,2) of our* TRACKAPPROXTOPK *problem, using a distinct-count sketch synopsis $\mathcal{X}$ with a total space requirement of $O(rs \log^2 m \log n)$, where $r = \Theta(\log \frac{n}{\delta})$ and $s = \Theta(\frac{U \log((n+\log m)/\delta)}{f_{v_k} \epsilon^2})$. The update time for $\mathcal{X}$ (per streaming flow update) is $O(r \log m)$, whereas the query time (to recover the top-k destinations) is $O(rs \log^2 m)$.* ∎

5 Time-Efficient Top-k Tracking

Our basic distinct-count sketch synopsis and BaseTopk estimator, while efficient in terms of space usage and update time for processing stream updates, can incur a high overhead for producing the top-k frequency estimates and corresponding destination addresses. By Theorem 4.4, the top-k query time for a distinct-count sketch is $O(rs \log^2 m)$ where, even though r is logarithmic in n, s can be fairly large (in the order of $\frac{U \log(n/\delta)}{f_{v_k} \epsilon^2}$) and, clearly, not poly-logarithmic in n, m. Thus, while our baseline distinct-count sketch estimation scheme may prove useful in environments with rapid stream updates and relatively-infrequent top-k estimation queries, it is *not* an effective tracking solution — its high query-time requirements render it unsuitable for environments that require the top-k destinations to be continuously tracked either for every single update or for every constant (small) number of updates.

In this section, we propose an effective *tracking* solution for our TRACKAPPROXTOPK problem based on an enhancement of our basic distinct-count sketch stream synopsis, termed a *Tracking Distinct-Count Sketch* (or, *Tracking-DCS*, for short). In a nutshell, the key idea behind our Tracking-DCS synopsis and top-k tracking scheme is to *incrementally maintain* the underlying distinct sample and corresponding destination occurrence frequencies over the stream of flow updates, rather than having to re-compute everything from the distinct-count sketch on every top-k estimation (as in BaseTopk). Our Tracking-DCS-based top-k estimation algorithm offers *guaranteed poly-logarithmic* update *and* query times, while increasing the overall storage space by only a small constant factor over our baseline distinct-count sketch synopsis.

The Tracking Distinct-Count Sketch Data Structure. Compared to a basic distinct-count sketch (Figure 2), a Tracking-DCS basically maintains some additional information for each bucket in the first-level hash table. More specifically, for each first-level bucket b, in addition to the r independent s-bucket second-level hash tables and count signatures, a

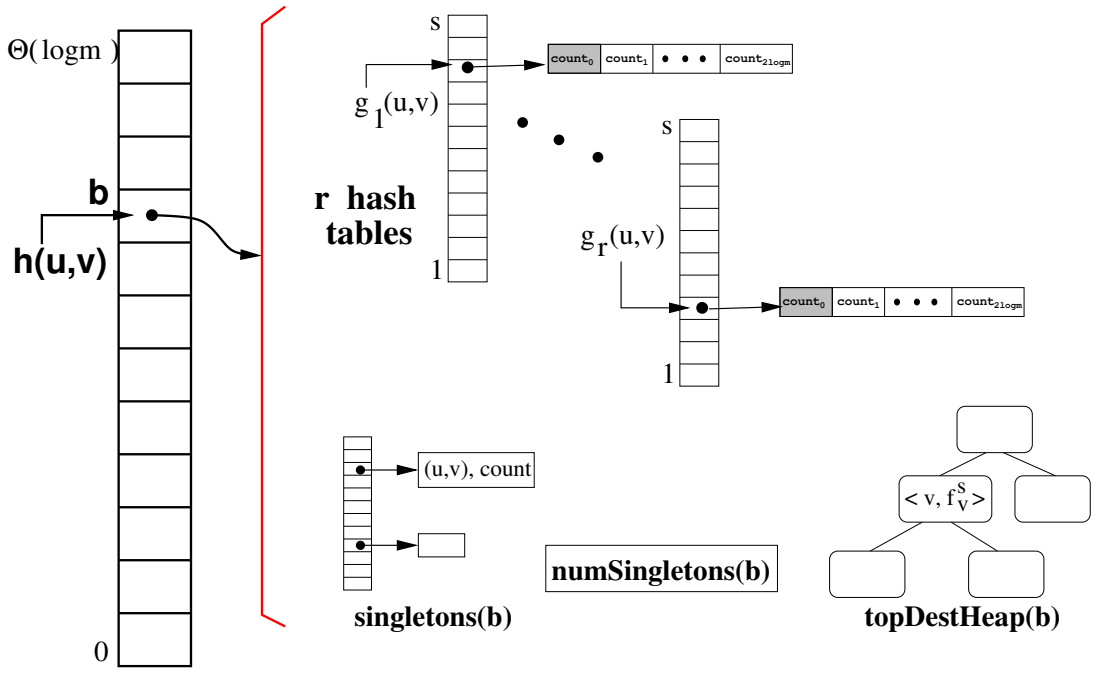

Figure 5: Our Tracking Distinct-Count Sketch Synopsis.

Tracking-DCS also stores:

1. *The current set* singletons(b) of singleton (source, destination) address pairs in bucket b (note, of course, that these pairs are exactly the distinct sample contributed by bucket b);

2. *A counter* numSingletons(b) recording the current number of singleton (source, destination) address pairs in bucket b (i.e., numSingletons$(b) = |$singletons$(b)|$); and,

3. *A heap* topDestHeap(b) containing all destinations v appearing in singleton address pairs in the distinct sample collected from first-level buckets $\geq b$ (i.e., $\cup_{l \geq b}$singletons(l)). topDestHeap(b) is organized as a max-heap according to the occurrence frequency f_v^s of destination v in the distinct sample $\cup_{l \geq b}$singletons(l), that is, $f_v^s = |\{(s,d) \in \cup_{l \geq b}\text{singletons}(l) : d = v\}|$.

Each entry in the singletons(b) set comprises a (singleton) address pair (u, v) in bucket b and a count recording the number of second-level hash tables where (u, v) appears as a singleton. The interface to the singletons(b) set data structure supports three key operations: (1) getCount(u, v): Returns the count for the (u, v) pair in singletons(b) (or, zero if the $(u, v) \notin$ singletons(b)); (2) incrCount(u, v): Increments the count for the (u, v) pair in singletons(b) (inserting that pair with a count of 1 if it is not already there); and, (3) decrCount(u, v): Decrements the count for the (u, v) pair

in singletons(b) (deleting the (u, v) entry if its count drops to zero). The key observation here is that, by virtue of the second-level structures in a distinct-count sketch, the number of observed singleton pairs in a first-level bucket is at most $r \cdot s$. Thus, a simple way of implementing singletons(b) is using a hash-table structure (on source-destination pairs) with $\Theta(rs)$ entries — this also allows us to implement each of the above interface operations for the singletons(b) set in $O(1)$ (expected) time. Similarly, note that the number of entries in each topDestHeap(b) data structure is at most $r \cdot s \cdot \Theta(\log m)$. Thus, it is easy to see that the space overhead of a Tracking-DCS is only a small constant factor over that of a basic distinct-count sketch synopsis (for the same r, s parameters). The structure of our Tracking-DCS synopsis is depicted in Figure 5.

Procedure UpdateTracking($\mathcal{TX}$, (u, v, Δ))
Input: Tracking Distinct-Count Sketch synopsis $\mathcal{TX}$ over input stream(s),
source-destination pair (u, v) with update action $\Delta = \pm 1$ (insert or delete).
Output: Updated Tracking Distinct-Count Sketch synopsis $\mathcal{TX}$.
begin
1. b := $h(u, v)$
2. **if** ($\Delta = +1$) **then** // insertion of the (u, v) address pair
3. **for** $j := 1$ **to** r **do** {
4. **if** (bucket $g_j(u, v)$ is a singleton **and** the address pair in $g_j(u, v)$ is $(u', v') \neq (u, v)$) **then**
5. { // bucket $g_j(u, v)$ no longer a singleton
6. decrCount(singletons(b), (u', v'))
7. **if** (getCount(singletons(b), (u', v')) = 0) **then**
8. { // (u', v') no longer in singletons(b) – update singleton counter and destination heaps
9. numSingletons(b) := numSingletons(b) -1
10. **for** $l := b$ **downto** 0 **do**
11. find entry for destination v' in topDestHeap(l), update frequency $f^s_{v'} := f^s_{v'} - 1$, and adjust the heap
12. }
13. }
14. **else if** (bucket $g_j(u, v)$ is empty) **then**
15. { // bucket $g_j(u, v)$ is a new singleton bucket
16. incrCount(singletons(b), (u, v))
17. **if** (getCount(singletons(b), (u, v)) = 1) **then**
18. { // new singleton occurrence – update counter, heaps
19. numSingletons(b) := numSingletons(b) $+1$
20. **for** $l := b$ **downto** 0 **do**
21. find entry for destination v in topDestHeap(l) (or, create one with $f^s_v = 0$ if not already there),
update frequency $f^s_v := f^s_v + 1$, and adjust the heap
22. }
23. }
24. insert (u, v) in the count signature for bucket $g_j(u, v)$
25. } // for
26. **else if** ($\Delta = -1$) **then** // deletion of the (u, v) address pair
27. // symmetric to the insertion case –
// possible second-level bucket transitions of interest: (non-singleton $\rightarrow$ singleton) and (singleton $\rightarrow$ empty).
28. $\cdots$
end

Figure 6: Updating a Tracking Distinct-Count Sketch Synopsis.

Maintenance. The algorithm for maintaining a Tracking-DCS synopsis $\mathcal{TX}$ over stream(s) of flow updates is slightly more complicated than that for a basic distinct-count sketch. The added complexity, of course, comes from maintaining the additional information in each first-level hash bucket up-to-date; as we will see, however, this extra complexity has little effect on the worst-case update time which remains small.

Our procedure for processing each streaming flow update over a Tracking-DCS synopsis $\mathcal{TX}$ is depicted in Figure 6. Briefly, for every stream update $(u, v, \Delta = \pm 1)$, our UpdateTracking procedure finds the appropriate first-level bucket $b = h(u, v)$ and updates the count signatures for each corresponding second-level bucket $g_j(u, v)$, $j = 1, \ldots, r$ (as in the case of a basic distinct-count sketch). Furthermore, for each of these r count-signature updates, UpdateTracking also keeps track of the status of the current distinct sample in bucket b (i.e., the singletons(b) set) as well as the corresponding singletons' counter (numSingletons(b)) and destination max-heaps (topDestHeap). Consider the case of an insertion of the (u, v) pair ($\Delta = +1$). This insertion can cause a second-level bucket $g_j(u, v)$ to transition from *singleton to non-singleton* (i.e., ≥ 2 bucket elements) (Steps 4-14), or from *empty to singleton* (Steps 15-24). (Other possible bucket transitions can have no effect on the distinct sample collected from bucket b.) In each case, the count for the affected singleton address pair in singletons(b) is appropriately updated and, in the case that the singleton is either deleted from (Steps 7-13) or newly-inserted in (Steps 18-23) singletons(b), the numSingletons(b) counter is adjusted and the topDestHeap(l) structures for all first-level buckets $l \leq b$ are updated to reflect the new frequency for the destination in the affected pair. The procedure for the case of a deletion ($\Delta = -1$) is completely symmetric to the insertion case, and not shown in detail in Figure 6; of course, in this case, the second-level bucket transitions of interest are *non-singleton to singleton* and *singleton to empty*.

In terms of update-time complexity, note that all the extra operations for maintaining the added distinct-sample tracking information in a Tracking-DCS have $O(1)$ time complexity, with the sole exception of the $b + 1 = O(\log m)$ topDestHeap adjustments which, since there can clearly be no more than m distinct destinations, can be done in $O(\log^2 m)$ time. (Updating the count-signature information (Step 23), as described in Section 3, is an $O(\log m)$ operation.) Thus, the (worst-case) maintenance time per streaming flow update for our Tracking-DCS synopsis is only $O(r \log^2 m)$.

Top-k Estimation. Incrementally maintaining the distinct-sample information in each first-level bucket of a Tracking-DCS (as described above), makes the estimation procedure for recovering the (approximate) top-k frequencies (and corresponding destination addresses) very simple. Our top-k tracking procedure, termed TrackTopk (depicted in Figure 7), essentially mimics the logic of our baseline BaseTopk estimator while exploiting (a) the per-bucket numSingletons

Procedure TrackTopk($\mathcal{TX}, \epsilon$)
Input: Tracking Distinct-Count Sketch synopsis $\mathcal{TX}$ over input stream(s), relative accuracy parameter ϵ.
Output: Approximate top-k destinations and corresponding distinct-source frequencies.
begin
1. // find the right distinct-sample inference level
2. b := index of topmost first-level hash bucket of $\mathcal{TX}$ // $b = \Theta(\log m)$
3. sampleSize := 0
4. **while** ($b \geq 0$ **and** sampleSize $< (1+\epsilon)s/16$) **do** {
5. sampleSize := sampleSize +numSingletons(b)
6. $b := b - 1$
7. }
8. // find top-k destinations using the heap at bucket level b
9. **for** $i := 1$ **to** k **do** {
10. // remove current maximum sample frequency
11. $< v_i, f^s_{v_i} >$:= deleteMax(topDestHeap(b))
12.}
13// scale obtained sample frequencies by 2^b
14**return** $\{< v_i\ ,\ 2^b \cdot f^s_{v_i} >:\ 1 \leq i \leq k\}$
end

Figure 7: Tracking Top-k Destinations using a Tracking-DCS.

counters to quickly infer the correct sample-inference level b (Steps 1-7), and (b) the topDestHeap structure in the chosen first-level bucket b to quickly recover the top-k destinations (without re-building the distinct sample from scratch) (Steps 8-12). Since each deleteMax heap operation (Step 11) for recovering the destination with the maximum occurrence frequency in the distinct sample has a time cost of at most $O(\log m)$, it is easy to see that the overall time complexity of our TrackTopk procedure for recovering the top-k destination frequencies is only $O(k \log m)$.

Theorem 5.1 summarizes the results of our analysis, identifying the space and update/query-time requirements for our tracking TrackTopk estimation procedure; the proof follows easily from Theorem 4.4 and our earlier discussion in this section.

Theorem 5.1 *Let $\epsilon < 1/3$. Procedure* TrackTopk *returns a list of (ϵ, δ) frequency estimates satisfying Clauses (1,2) of our* TRACKAPPROXTOPK *problem, using a distinct-count sketch synopsis $\mathcal{TX}$ with a total space requirement of $O(rs \log^2 m \log n)$, where $r = \Theta(\log \frac{n}{\delta})$ and $s = \Theta(\frac{U \log((n+\log m)/\delta)}{f_{v_k} \epsilon^2})$. The update time for $\mathcal{TX}$ (per streaming flow update) is $O(r \log^2 m)$, whereas the query time (to recover the top-k destinations) is $O(k \log m)$.* ∎

We summarize the (worst-case) space/time requirements for our two proposed families of Distinct-Count Sketch synopses (Basic and Tracking) in Table 2.

	Basic DCS	**Tracking DCS**
Space	$\Theta(\frac{U \log^3 \frac{n}{\delta} \log^2 m}{f_{v_k} \epsilon^2})$	$\Theta(\frac{U \log^3 \frac{n}{\delta} \log^2 m}{f_{v_k} \epsilon^2})$
Update Time	$O(\log \frac{n}{\delta} \log m)$	$O(\log \frac{n}{\delta} \log^2 m)$
Query Time	$O(\frac{U \log^2 \frac{n}{\delta} \log^2 m}{f_{v_k} \epsilon^2})$	$O(k \log m)$

Table 2: Basic vs. Tracking Distinct-Count Sketch.

6 Experimental Study

In this section, we present preliminary results from an experimental study that we have conducted to verify the effectiveness of our sketching synopses and techniques for tracking the top-k destination frequencies in small space/time. The major findings of our study can be summarized as follows.

- **Effective Stream Summarization for Top-k Distinct-Source Frequency Queries.** By maintaining small distinct-count sketch synopses over the stream(s) of flow updates our estimation algorithms are able to recover the top-k destinations (and, corresponding distinct-source frequencies) quite accurately. More specifically, using a distinct-count sketch summary whose size is only a small fraction of the total number U of distinct source-destination address pairs in the stream, we are able to answer top-k frequency queries with high recall and low relative estimation error.

- **Small Synopsis Update and Top-k Query Times.** Both our basic and tracking distinct-count sketch maintenance algorithms incur a small processing overhead per stream update that is of the order of a few tens of microseconds. Furthermore, our Tracking-DCS synopses can handle continuous top-k query tracking with minimal overhead, whereas the performance of the basic distinct-count sketch can suffer for high query frequencies.

Thus, our experimental results validate our approach, demonstrating that our proposed distinct-count sketch synopses are indeed viable, effective tools for detecting destinations in large ISP networks that may be under a DDoS attack (in real time). All experiments reported in this paper were run on a 1GHz Intel Pentium-III machine with 256MB of main memory, running Redhat Linux 7.2.

6.1 Experimental Testbed and Methodology

Stream-Synopsis Implementation. We implemented both the Basic and the Tracking versions of our distinct-count sketch stream synopses and top-k estimation algorithms (described in Sections 3-5). We varied the two key parameters of our implemented distinct-count sketch synopses, namely the number of inner hash tables r and the number of buckets

per inner hash table s between 3–4 and 64–256, respectively. The default values used in the bulk of our experimental runs were $r = 3$ and $s = 128$; the numbers reported below are indicative of the results obtained for other parameter settings.

Data Sets. We employed a synthetic data generator based on Zipfian frequency distributions [37] (with various levels of skew) to produce our stream of source-destination address-pair updates. Our update-stream generation process is characterized by three key parameters: the total number of distinct source-destination IP-address pairs U, the number of distinct destinations d, and the Zipfian skew parameter z that determines the distribution of distinct source IP addresses in our input stream across the d distinct destinations. We varied the U parameter between $2 \cdot 10^6$–$16 \cdot 10^6$, the d parameter between 10^3–10^5, and the z parameter between 1 (moderate skew) and 2.5 (extreme skew). We believe that such moderate-to-high frequency-skew values are indicative of real-life DDoS attack scenarios, where a large part of the total "mass" of observed pairs U is concentrated to only a few of the d distinct destination addresses in the underlying ISP network. The default parameter settings used in most of our experiments were $U = 8 \cdot 10^6$ and $d = 5 \cdot 10^4$; once again, qualitatively similar numbers were obtained for other settings.

Distinct-Count Sketch Synopsis Space Requirements. Assuming the default number of distinct source-destination pairs $U = 8 \cdot 10^6$, the number of non-empty first-level hash buckets in our constructed distinct-count sketch synopses is approximately equal to 23 – this is because $8 \cdot 10^6 \approx 2^{23}$ and the probability of a pair mapping to level i drops exponentially with i (Section 3). Thus, since each count-signature array maintains $(64 + 1) = 65$ integer counters (each 4 bytes long), and assuming our default distinct-count sketch parameter values of $r = 3$ and $s = 128$, the total space requirements (in bytes) of our Basic distinct-count sketch synopsis in this setting are $23\times\ 3\times\ 128\times\ 65\times\ 4 \approx$ 2.3MB. With its extra data structures, the Tracking version of our distinct-count sketch increases this space requirement by a factor of about two, yielding a total synopsis size of approximately 4.6MB. On the other hand, note that a naive, "brute-force" scheme for maintaining distinct-source frequencies over a stream of flow updates with $U = 8 \cdot 10^6$ distinct source-destination pairs would require approximately 96MB of space. This is essentially the space needed to store the source and destination IP addresses (4 bytes per address) as well as frequency counts (4 bytes per count) for the observed 8 million source-destination pairs. Thus, our sketching techniques offer storage-space benefits of well over an order of magnitude in this setting.

Of course, the space benefits of our distinct-count sketch synopses become much more impressive as the number of distinct pairs U grows. For example, when U is increased from $8{\cdot}10^6$ to $10^9 \approx 2^{30}$, the space requirements of our sketches grow by a factor of about $30/23 = 1.3$ (i.e., approximately 30 instead of 23 non-empty first-level buckets), yielding a total size of about 6MB for a Tracking Distinct-Count Sketch. In contrast, the space needed to store all of U (and corresponding frequency counters) explodes by a factor of $2^{30}/2^{23} = 128$ for a total space requirement of over 12GB,

giving our sketches a storage gain of over three orders of magnitude! Coupled with the observed small update/query times and the excellent result accuracy of our sketches for top-k queries, the above analysis clearly demonstrates the effectiveness of our solution for real-time tracking of DDoS activity over massive ISP networks.

Performance Metrics. We employed two key metrics to gauge the effectiveness of our sketch-based estimation techniques for top-k queries. The first metric is the *top-k recall* defined as the fraction of the true top-k destinations in the approximate top-k result returned by our estimators based on the distinct-count sketch synopsis (for various values of k). The second metric is the *average relative error* in the distinct-source frequency estimates $\hat{f}_v$ returned by our estimators for the true top-k destinations v found in the approximate answer; that is, the quantity $\sum_{v \in R} \frac{|\hat{f}_v - f_v|}{f_v}$, where R denotes the "recall set" for the top-k query ($|R| \leq k$). (Note, of course, that these accuracy numbers are independent of the specific version (i.e., Basic or Tracking) of distinct-count sketch used.) Finally, to gauge the effectiveness of our techniques in dealing with high-speed data streams, we measured the *processing time per stream update* for both Basic and Tracking distinct-count sketch synopses in top-k tracking environments with different mixes of stream updates and top-k queries. To account for the randomness in our techniques, all numbers reported below represent averages over 5 runs of our algorithms with different random seeds.

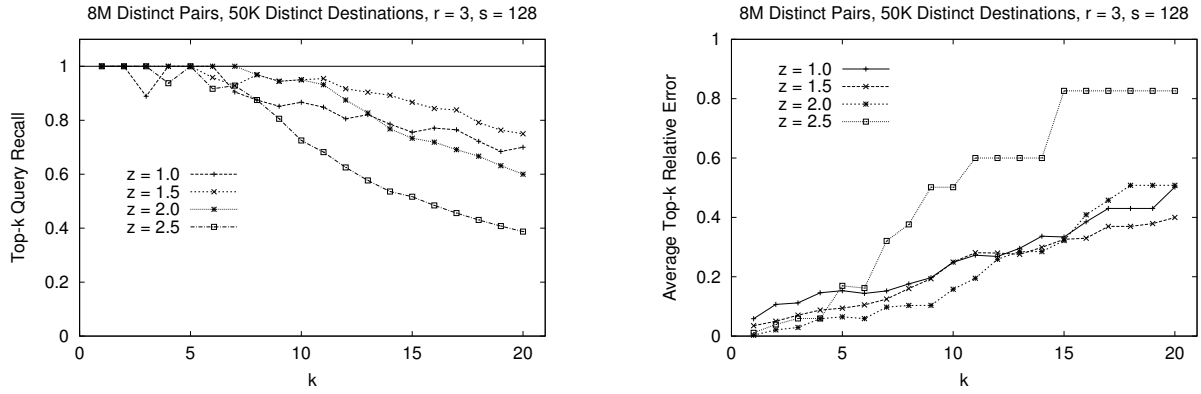

Figure 8: Top-k Estimation Accuracy Results: (a) Top-k Query Recall. (b) Average Relative Error in Top-k Frequencies.

6.2 Experimental Results

Top-k Estimation Accuracy. Figures 8(a,b) depict the top-k recall and average relative error numbers obtained with our estimation algorithms (as a function of k), using distinct-count sketch synopses (with the default parameters of $r = 3$ and $s = 128$) over a stream of flow updates with 8 million distinct source-destination pairs and $50,000$ distinct destinations for different values of the Zipfian skew parameter z. Clearly, our sketching-based estimators are able to

correctly identify the top few destinations in the update stream with remarkable accuracy; for example, the recall for the top-k destinations with $k \leq 5$ is almost always 100% for all skew values z. (The observed dips in recall are basically due to our estimators occasionally reversing the order of neighboring top-k elements.) As expected, for larger values of k the recall of our approximate top-k results drops but still remains reasonably high; for example, for all skew values $z \leq 2.0$, our estimators retrieve more than 86% of the top-k destinations for all $k \leq 10$, and more than 73% for $k \leq 15$. The drop in top-k recall is obviously much more dramatic for the extreme skew parameter value of $z = 2.5$; this is natural since, with such extreme skew, more than 95% of the distinct-source frequency mass is concentrated in the top-5 destinations (which are typically correctly identified by our estimators), with remaining destinations having significantly smaller observed frequencies.

Similar trends can be observed in the average relative error plots of Figure 8(b): the relative error of the approximate frequencies returned by our distinct-count sketch algorithms is less than 17% for the top-5 destinations (for all skew values z) and, for $z \leq 2.0$, the estimation error is upper-bounded by 25% and 34% for the top-10 and top-15 destinations, respectively. Once again, as expected, the increase in frequency estimation error with larger values of k is much more dramatic for extreme skew values.

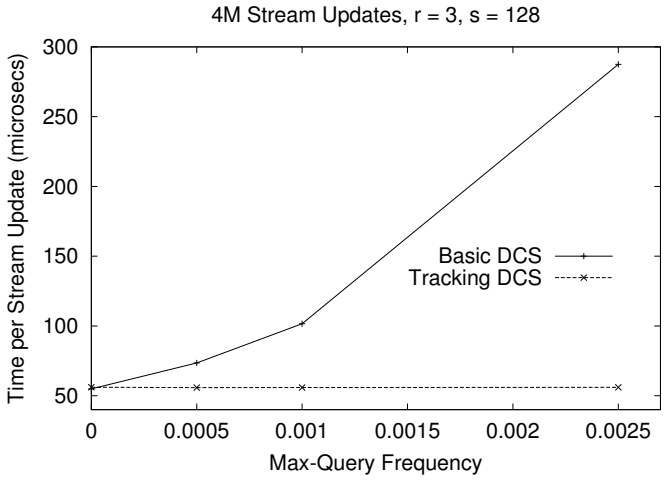

Figure 9: Per-Update Processing Times (in μsec).

Processing-Time Overheads. We found the update algorithms for both our Basic and Tracking distinct-count sketch synopses to be extremely fast, typically taking between 40–60 microseconds (μsecs) to process each stream update. (We believe that further optimizations in our code can reduce these numbers even further.) To differentiate between Basic and Tracking sketches and demonstrate the effectiveness of our Tracking distinct-count sketch in continuously tracking the

top-k frequencies, we employed a stream of *max* (i.e., top-1) queries in parallel with the stream of flow updates, varying the query-occurrence frequency (i.e., the max-element tracking frequency) in the stream. Figure 9 depicts the observed average processing time per update for a stream of 4 million flow updates as the max-query frequency is varied between 0 (i.e., no queries) and 0.0025 (i.e., one max-query for every 400 stream updates). For a stream of pure updates (query frequency $= 0$) both Basic and Tracking sketches spend about 55–56 μsecs per update; as the query frequency grows, however, the observed average times per-update for the Tracking distinct-count sketch remain approximately constant, whereas the corresponding times for the Basic distinct-count sketch show a dramatic increase (due to its significantly more expensive BaseTopk estimator) reaching about 290 μsecs (for query frequency $= 0.0025$).

7 Conclusions

We have proposed novel data-streaming algorithms for the robust, real-time detection of DDoS activity in large ISP networks. The key to our solution lies in new synopsis data structures that allow us to efficiently *track* (in guaranteed small space and time) the top distinct-source frequencies over a general stream of updates to the set of underlying network flows. Our experimental study has demonstrated the effectiveness of our approach, showing that our algorithms accurately track large distinct frequencies in small space and guaranteed small update/query time.

References

[1] "Defeating DDoS Attacks". Riverhead Networks White Paper (`http://www.riverhead.com/`).

[2] "NetFlow Services and Applications". Cisco Systems White Paper (`http://www.cisco.com/`), 1999.

[3] "CERT Advisory CA-1996-21: TCP SYN Flooding and IP Spoofing Attacks". (`http://www.cert.org/`), Nov. 2000.

[4] A. Akella, A. Bharambe, M. Reiter, and S. Seshan. "Detecting DDoS Attacks on ISP Networks". In *MPDS*, June 2003.

[5] N. Alon, Y. Matias, and M. Szegedy. "The Space Complexity of Approximating the Frequency Moments". In *ACM STOC*, 1996.

[6] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". *Comm. of the ACM*, 13(7), July 1970.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *"Introduction to Algorithms"*. MIT Press, 1990.

[8] G. Cormode, and S. Muthukrishnan. "Space Efficient Mining of Multigraph Streams". In *ACM PODS*, 2005.

[9] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. "Gigascope: A Stream Database for Network Applications". In *ACM SIGMOD*, June 2003.

[10] C. Estan and G. Varghese. "New Directions in Traffic Measurement and Accounting". In *ACM SIGCOMM*, Aug. 2002.

[11] C. Estan, G. Varghese, and M. Fisk. "Bitmap Algorithms for Counting Active Flows on High-Speed Links". In *ACM SIGCOMM Internet Measurement Workshop*, 2003.

[12] P. Flajolet and G. N. Martin. "Probabilistic Counting Algorithms for Data Base Applications". *Journal of Computer and Systems Sciences*, 31:182–209, 1985.

[13] S. Ganguly, M. Garofalakis, and R. Rastogi. "Processing Set Expressions over Continuous Update Streams". In *ACM SIGMOD*, 2003.

[14] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani. "Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks". Bell Labs Tech. Memo.

[15] Y. Gao, Z. Li, and Y. Chen. "A DoS Resilient Flow-Level Intrusion Detection Approach for High-Speed Networks". In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[16] M. Garofalakis, J. Gehrke, and R. Rastogi. "Querying and Mining Data Streams: You Only Get One Look". Tutorial in *VLDB*, 2002.

[17] P. B. Gibbons and Y. Matias. "New Sampling-Based Summary Statistics for Improving Approximate Query Answers". In *ACM SIGMOD*, June 1998.

[18] P. B. Gibbons. "Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports". In *VLDB*, Sep. 2001.

[19] P. B. Gibbons and S. Tirthapura. "Estimating Simple Functions on the Union of Data Streams". In *ACM SPAA*, July 2001.

[20] A. Hussain, J. Heidemann, and C. Papadopoulos. "A Framework for Classifying Denial-of-Service Attacks". In *ACM SIGCOMM*, 2003.

[21] J. Jung, B. Krishnamurthy, and M. Rabinovich. "Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites". In *WWW*, May 2002.

[22] R. Kompella, S. Singh, and G. Varghese. "On Scalable Attack Detection in the Network". In *ACM SIGCOMM IMC*, 2004.

[23] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. "Sketch-Based Change Detection: Methods, Evaluation, and Applications". In *ACM SIGCOMM IMC*, 2003.

[24] A. Kuzmanovic and E. Knightly. "Low-Rate TCP-Targeted Denial-of-Service Attacks". In *ACM SIGCOMM*, 2003.

[25] G. S. Manku and R. Motwani. "Approximate Frequency Counts over Data Streams". In *VLDB*, Aug. 2002.

[26] D. Moore, G. M. Voelker, and S. Savage. "Inferring Internet Denial-of-Service Activity". In *USENIX*, 2001.

[27] D. Moore, C. Sharon, D. Brown, G. M. Voelker, and S. Savage. "Inferring Internet Denial-of-Service Activity". In *ACM TOCS*, 2006.

[28] R. Motwani and P. Raghavan. *"Randomized Algorithms"*. Cambridge University Press, 1995.

[29] V. Paxson. "An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks". In *SIGCOMM Computer Comm. Review*, 31(3), July 2001.

[30] V. Paxson. "Bro: A System for Detecting Network Intruders in Real-time". In *Computer Networks*, 31(23-24), Dec 1999.

[31] M. Roesch. "Snort - Lightweight Intrusion Detection for Networks". In *USENIX*, 1999.

[32] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. "New Streaming Algorithms for Superspreader Detection". In *Network and Distributed System Security Symposium NDSS*, 2005.

[33] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker "DDoS Defense by Offense". In *ACM SIGCOMM*, 2006.

[34] N. Weaver, S. Staniford, and V. Paxson. "Very Fast Containment of Scanning Worms". In *USENIX*, 2004.

[35] X. Yang, D. Weatherall, and T. Anderson. "A DoS-Limiting Network Architecture". In *ACM SIGCOMM*, 2005.

[36] H. Wang, D. Zhang, and K. G. Shin. "Detecting SYN Flooding Attacks". In *IEEE INFOCOM*, June 2002.

[37] G. K. Zipf. *"Human Behavior and the Principle of Least Effort – An Introduction to Human Ecology"*. Addison-Wesley Press, 1949.