New Streaming Algorithms for Fast Detection of Superspreaders

Shobha Venkataraman[1] Dawn Song[1]
Phillip B. Gibbons[2] Avrim Blum[1]

May 2004

CMU-CS-04-142 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1] Computer Science Dept, Carnegie Mellon University. Email: {shobha@cs., dawnsong@, avrim@cs.}cmu.edu
[2] Intel Research Pittsburgh. Email: phillip.b.gibbons@intel.com

This research was supported in part by NSF-ITR CCR-0122581 and the Center for Computer and Communications Security at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of NSF, ARO, Carnegie Mellon University, or the U.S. Government or any of its agencies.

Keywords: network monitoring, streaming algorithms, anomaly detection, heavy distinct-hitters, superspreaders

Abstract

High-speed monitoring of Internet traffic is an important and challenging problem, with applications to real-time attack detection and mitigation, traffic engineering, etc. However, packet-level monitoring requires fast streaming algorithms that use very little memory space and little communication among collaborating network monitoring points.

In this paper, we consider the problem of detecting *superspreaders*, which are sources that connect to a large number of *distinct* destinations. We propose several new streaming algorithms for detecting superspreaders, and prove guarantees on their accuracy and memory requirements. We also show experimental results on real network traces. Our algorithms are substantially more efficient (both theoretically and experimentally) than previous approaches. We also provide several extensions to our algorithms – we show how to identify superspreaders in a distributed setting, with sliding windows, and when deletions are allowed in the stream.

More generally, our algorithms are applicable to any problem that can be formulated as follows: given a stream of (x, y) pairs, find all the x's that are paired with a large number of distinct y's. We call this the *heavy distinct-hitters* problem. There are many network security applications of this general problem. This paper discusses these and other applications, and for concreteness, focuses on the superspreader problem.

1 Introduction

Internet attacks such as distributed denial-of-service (DDoS) attacks and worm attacks are increasing in severity. Network security monitoring can play an important role in defending and mitigating such large-scale Internet attacks - it can be used to detect drastic traffic pattern changes that may indicate attacks, or more actively, to identify misbehaving hosts or victims being attacked, and to develop appropriate filters to throttle attack traffic automatically.

For example, a compromised host doing fast scanning for worm propagation often makes an unusually high number of connections to distinct destinations within a short time. The Slammer worm, for instance, caused some infected hosts to send up to 26,000 scans a second [26]. We call such a host a *superspreader.* (Note that a superspreader may also be known as a port scanner in certain cases.) By identifying in real-time any source IP address that makes an unusually high number of distinct connections within a short time, a network monitoring point can identify hosts that may be superspreaders, and take appropriate action. For example, the identified potential attackers (and victims) can be used to trigger the network logging system to log attacker traffic for detailed real-time and post-mortem analysis of attacks (including payload analysis), and can also be used to help develop filters that throttle subsequent (similar) attack traffic in real-time.

In addition to network monitoring at a single point, distributed network security monitoring, where network monitoring points distributed throughout the network collaborate and share information, can contribute additional benefits and enhance the monitoring capability. Distributed network monitoring collectively covers more traffic, and can often identify distributed attacks faster than any single monitoring point can.

Network security monitoring, however, has many challenges and requires efficient algorithms. More and more high speed links are being deployed. The traffic volume on high speed links can be tens of gigabits per second, and can contain millions of flows per minute. In addition, within such a great number of flows and high volume of traffic, most of the flows may be normal flows. How does one find the needle in the haystack efficiently? That is, how does one find attack traffic in the huge volume of normal traffic efficiently? Many traditional approaches require the network monitoring points to maintain per-flow state. Keeping per-flow state, however, often requires high memory storage, and hence is not practical for high speed links. Because DRAM is too slow for line speed access on high speed links, network monitoring points would need to use SRAM to keep state for monitoring; however, SRAM is very expensive, and hence any monitoring algorithm should use memory sparingly.

Moreover, distributed network monitoring poses additional challenges - because distributed network monitoring relies on the communication among network monitoring points, the distributed monitoring algorithm needs to have low communication overhead among cooperating peers to avoid overburdening monitoring points with update traffic.

In this paper, we propose several new efficient algorithms for fast network security monitoring and distributed network security monitoring. In particular, we present new efficient algorithms for identifying *super spreaders.* We provide both theoretical analysis and experimental results of our new algorithms, and demonstrate that these algorithms have important applications. Our algorithms are substantially more efficient (both theoretically and experimentally) than previous approaches, and have additional benefits. Our analysis yields tight accuracy and performance bounds on our algorithms.

Note that a superspreader is different from the usual definition of a heavy-hitter ([18, 6, 15, 25, 11, 22]). A heavy-hitter might be a source that sends a lot of packets, and thus exceeds a certain threshold of the total traffic. A superspreader, on the other hand, is a source that contacts many *distinct* destinations. So, for instance, a source that is involved in a few extremely large file (si, dl), (s2, d2), (si, dl), (53, d3), (si, dl), (52, d3), (54, dl), (52, d4), (si, dl), (55, d4), (56, d6)

Figure 1: Example stream of (source, destination) pairs, starting with (sl,dl) and ending with (56,d6).

transfers may be a heavy-hitter, but is not a superspreader. On the other hand, a source that sends a single packet to many destinations might not create enough traffic to be a heavy-hitter, even if it is a superspreader - some of the sources in our traces that are superspreaders create less than 0.004% of the total traffic analyzed; heavy-hitters typically involve a significantly higher fraction of the traffic.

The superspreader problem is an instance of a more general problem that we term *heavy distinct-hitters,* which may be formulated as follows: given a stream of *(x,y)* pairs, find all the x's that are paired with a large number of distinct y's. Figure 1, for example, depicts a stream where source s2 is paired with three distinct destinations, whereas all other sources in the stream are paired with only one distinct destination; thus s2 is a heavy distinct-hitter for this (short) stream. We discuss other applications of heavy distinct-hitters in Section 2.

To summarize, this paper makes the following contributions:

- We propose new streaming algorithms for identifying *super spreaders.* Our algorithms are the first to address this problem efficiently and provide proven accuracy and performance bounds. The best previous approach [16] requires a certain amount of memory to be allocated for each source within the time window. Our algorithms are the first ones where the amount of memory allocated is independent of the number of sources in the time window, resulting in significant memory savings. In addition, one of our algorithms is based on a novel two-level sampling scheme, which may be of independent interest.

- We also propose several extensions to enhance our algorithms - we propose efficient distributed versions of our algorithms (Section 4.1), we extend our algorithms to scenarios when deletion is allowed in the stream (Section 4.2), and we extend our algorithms to the sliding window scenario (Section 4.3).

- Our experimental results on traces with up to 4.5 million packets confirm our theoretical results. Further, they show that the memory usage of our algorithms is substantially smaller than alternative approaches. Finally, they study the effect of different superspreader thresholds on the performance of the algorithms, again confirming the theoretical analysis.

Note that the contribution of this paper is in the proposal of new streaming algorithms to enable efficient network monitoring for attacks detection and defense, when *given* certain parameters. Selecting and testing the correct parameters, however, is application-dependent and out of the scope of this paper.

The rest of the paper is organized as follows. Section 2 motivates and defines the superspreader problem. Section 3 presents and compares two novel algorithms for the superspreader problem. Section 4 presents our extensions to handle distributed monitoring, deletions, and sliding windows. Section 5 presents our experimental results. Section 6 describes related work, and Section 7 presents conclusions.

2 Finding Superspreaders

In this section, we motivate the *superspreaders* problem, present a formal definition of the problem, and then discuss the deficiencies of previous techniques in addressing the problem.

2.1 Motivation and Applications

During fast scanning worm propagation, a compromised host may try to connect to a high number of distinct hosts in order to spread the worm; we call such a host a *superspreader.* Superspreaders could be responsible for contacting a lot of destinations early during worm propagation; so, detecting them early is of paramount importance. Thus, given a sequence of packets, we would like to design an efficient monitoring algorithm to identify in real-time which source IP addresses have contacted a high number of distinct hosts within a time window. We call this problem the *superspreader* problem. Note that if a source makes multiple connections or sends multiple packets to the *same* destination within the time window, the source-destination connection will be counted only once. This is necessary in order to avoid flagging the common legitimate communication patterns where a source either makes several connections to the same destination within a time window (such as with webpage downloads) or sends many packets to the same destination (such as when sending a large file).

It is important to develop efficient mechanisms to identify superspreaders on high-speed networks. For example, for a large enterprise network or an ISP network for a large number of home users, identifying such superspreaders in real-time would aid in throttling attack traffic, and help system administrators to find the computers that are compromised and need to be isolated and cleaned-up. This is a difficult problem on a high-speed monitoring point, as there may be millions of legitimate flows passing by per minute and the attack traffic may be an extremely small portion of the total traffic.

Extensions We also consider several extensions to the superspreader problem. First, we consider the distributed version of the superspreader problem. In the distributed setting, distributed network monitoring points collectively identify superspreaders in the overall traffic.

In the second extension, we consider the superspreader problem with *deletion.* For example, instead of identifying a source IP address that contacts a high number of distinct destinations within a time window, we may wish to identify a source IP address that contacts a number of distinct destinations and did not get a legitimate response from a high number of those contacted destinations (this may indicate a scanning behavior). Thus, once the network monitoring point sees a response from a destination for a connection from a source, the source-destination connection will be deleted from the count of the number of distinct connections a source makes.

In the third extension, we consider the superspreader problem over a *sliding window* of the most recent packets, e.g., the million most recent packets or the packets arriving in the last 60 minutes. The goal is to use far less space than would be needed to store all the packets in the window.

Other Applications Beyond identifying superspreaders, the techniques we propose solve a more general problem, which we call the *heavy-distinct-hitters* problem. In its abstract formulation, we have a stream of (#, *y)* pairs, and we want to identify the values of *x* occur in pairs with a large number of *distinct y* values. (For example, in the superspreader problem, *x* is the source IP and *y* is the destination IP.) The heavy-distinct-hitters problem has a wide range of applications. It can be used to identify which port has a high number of distinct destinations or distinct source-destination pairs without keeping per-port information and thus aid in detection of attacks such as worm propagation. Such a port is a heavy-distinct-hitter in our setting *(x* is the port and *y* is the destination or source-destination pair). Our technique can also be used to identify which port has high ICMP traffic which often indicates high scanning activity and scanning worm propagation, without keeping per-port information. The heavy distinct-hitters problem also has many networking applications. For example, spammers often send the same emails to many distinct destinations within a short period, and we could identify potential spammers without keeping information for every sender. For another example, in peer-to-peer networks, it could be used to find nodes that

N	Total no. of packets in a given time interval
t	A superspreader sends to more than *tN* distinct destinations
b	A false positive is a source that contacts fewer than *tN/b* distinct destinations but is reported as a superspreader
8	Probability that a given source becomes a false negative or a false positive
h *k*	$= t \cdot N$
W	Sliding window size
s	Source IP address
d	Destination IP address

Table 1: Summary of notation

talk to a lot of other nodes, without keeping per-node information.

For simplicity, in the rest of this section, we describe our algorithms for identifying superspreaders. The algorithms can be easily applied to the other security applications mentioned above.

2.2 Problem Definition

We define a *t-superspreader* as a host which contacts more than *t* • *N* unique destinations within a given window of *N* source-destination pairs. In Figure 1, for example, with *t* = |, source 52 is the only t-superspreader.[1] It follows from a lower bound in [1] that any deterministic algorithm that accurately estimates (e.g., within 10%) the number of unique destinations for a source needs *Cl(t - N)* space. Because we are interested in small space algorithms, we must consider instead randomized algorithms.

More formally, given a user-specified *b* > 1 and confidence level 0 < *5* < 1, we seek to report source IPs such that a source IP which contacts more than *tN* unique destination IPs is reported with probability at least 1 — *S* while a source IP with at most *tN/b* distinct destinations is (falsely) reported with probability at most *8.*

Table 1 summarizes the notation used in this paper.

2.3 Challenges and Previous Approaches

Identifying superspreaders efficiently on a high-speed network monitoring point is a challenging task. We now discuss existing approaches that may be applied to this problem, and their deficiencies.

- *Approach 1:* As a first approach, Snort [29] simply keeps track of each source and the set of distinct destinations it contacts within a specified time window. Thus, the memory required by Snort will be at least the total number of distinct source-destination pairs within the time window, which is impractical for high-speed networks.

- *Approach 2:* Instead of keeping a list of distinct destinations that a source contacts for each source, an improved approach may be to use a (randomized) distinct counting algorithm to

[1] Alternatively, a superspreader could be defined based on exceeding a given threshold *k* independent of *N*. The algorithms we present can readily be adapted to this case.

keep an approximate count of distinct destinations a source contacts to for each source [16]. (A *distinct counting algorithm* estimates the number of distinct elements in a data stream [1, 3, 4, 8, 10, 17, 19, 20, 16]). Along these lines, Estan et al. [16] propose using bitmaps to identify port-scans. The triggered bitmap construction that they propose keeps a small bitmap for each distinct source, and once the source contacts more than 4 distinct destinations, expands the size of the bitmap. Such an approach requires *n* • *S* space where *n* is the total number of distinct sources (which can be *Q(N))* and 5 is the amount of space required for the distinct counting algorithm to estimate its count. These approaches are particularly inefficient when the number of t-superspreaders is small and many sources contact much fewer than *t* destinations.

- *Approach 3:* Another approach that has not been previously considered is using a heavy-hitter algorithm in conjunction with a distinct-counting algorithm. We use a modified version of a heavy-hitter algorithm to identify sources that send to many destinations. Specifically, whereas heavy-hitters count the number of destinations, we count (approximately) the number of distinct destinations. This is done using a distinct counting algorithm. In our experiments we compare with this approach, with LossyCounting [25] as the heavy-hitter algorithm, and the first algorithm from [3] as the distinct-counting algorithm. The results show that our algorithms use much less memory than this approach; the details are in Section 5. For completeness, we summarize LossyCounting and the distinct counting algorithm we use in Appendix IV.

3 Algorithms for Finding Superspreaders

We propose new efficient algorithms where the memory storage space is independent of the size of the stream. In the remainder of this section, we describe two efficient algorithms for identifying superspreaders and evaluate their efficiency and accuracy. Algorithm I is simpler than Algorithm II; however, Algorithm II is more space-efficient for most cases. Further, Algorithm II presents a novel two-level sampling scheme, which may be of independent interest.

3.1 Algorithm I

The intuition for our first algorithm for identifying t-superspreaders over a given interval of *N* source-destination pairs is as follows.

We observe that if we sample the *distinct* source-destination pairs in the packets such that each distinct pair is included in the sample with probability p, then any source with *k* distinct destinations is expected to occur *kp* times in the sample. If *p* were ^, then any t-superspreader (with its $k \geq tN$ distinct destinations) would be expected to occur at least once in the sample, whereas sources that are not t-superspreaders would be expected *not* to occur in the sample. In this way, we may hope to use the sample to identify t-superspreaders.

There are several difficulties with this approach. First, the resulting sample would be a mixture of t-superspreaders and other sources that got "lucky" to be included in the sample. In fact, the expected size of the sample is independent of the number of t-superspreaders: If there are no *t*-superspreaders, for example, the sample will consist only of lucky sources. To overcome this, we set *p* to be a constant factor *c* larger than *j* ^. Then, any t-superspreader is expected to occur at least *c* times in the sample, whereas lucky sources may occur a few times in the sample but nowhere near *c* times. To minimize the space used by the algorithm, we seek to make *C* as small as possible while being sufficiently large to distinguish t-superspreaders from lucky sources.

A second, related difficulty is that there may be "unlucky" t-superspreaders that fail to appear in the sample as many times as expected. To overcome this, we have a second parameter r < *c* and report a source as a t-superspreader as long as it occurs at least r times in the sample. A careful choice of *c* and r are required.

Finally, we need an approach for uniform sampling from the *distinct* source-destination pairs. To accomplish this, we use a random hash function that maps source-destination pairs to [0,1) and include in the sample all distinct pairs that hash to [0,p). Thus each distinct pair has probability *p* of being included in the sample. Using a hash function ensures that the probability of being included in the sample is not influenced by how many times a particular pair occurs. On the other hand, if a pair is selected for the sample, then all its duplicate occurrences will also be selected. To fix this, our algorithm checks for these subsequent duplicates and discards them.

3.1.1 Algorithm Description

Let *srcIP* and *dstIP* be the source and destination IP addresses, respectively, in a packet. Let *hi* be a uniform random hash function that maps (srcIP, dstIP) pairs to [0,1), (that is, each input is equally likely to map to any value in [0,1) independently of other inputs). At a high level, the algorithm is as follows:

- Retain all distinct (srcIP, dstIP) pairs such that foi(srcIP, dstIP) < ^, where

  $$\mathrm{Cl} = \begin{cases} \ln(1/\delta)\left(\frac{3b+2b\sqrt{6b}+2b^2}{(b-1)^2}\right) & \text{if } 6 \leq 3 \\ \ln^*(1/\sqrt{\cdot})\text{W}^\wedge\text{V} & \\ \max(6,2/(1 - \%)^2) & \text{if } 3 < b < 2e^2 \\ 81n(l/5) & \text{if } 6 \geq 2e^2 \end{cases} \qquad (1)$$

  is determined by the analysis.

- Report all srcIPs with more than *r* retained, where

  $$r = \begin{cases} \frac{c_1}{b} + \sqrt{\frac{3c_1}{b}\ln(1/\delta)} & \text{if } 6 \leq 3 \\ {}^{\varsigma}f & \text{if } 3 < b < 2e^2 \\ \mathbf{f} & \textbf{if } b \geq \mathbf{2e^2} \end{cases} \qquad (^2)$$

Note that *c* is a modest constant when *b* and *5* are ©(1). Consider for example *S* = .05. When *b* = 2, *c* « 21. When 3 < *b* < 2e^2, *c* < 45, and in particular, *c* « 29 when 6 = 5 and *c* « 30 when *b* = 10. When *b* $\geq$ 2e^2, c_x « 24.

In more detail, the above steps can be implemented as follows. Our implementation has the desirable property that each t-superspreader is reported as soon as it is detected. We use two hash tables: one to detect and discard duplicate pairs from the sample, and the other to count the number of distinct destinations for each source in the sample. This latter hash table uses a second uniform random hash function *h^* that maps srcIPs to [0,1).

- *Initially:* Let *T* be a hash table with *c\/t* entries, where each entry contains an initially empty linked list of (srcIP, dstIP) pairs. Let T2 be a hash table with *c\/t* entries, where each entry contains an initially empty linked list of (srcIP, count) pairs.

- *On arrival of a packet with srcIP s and dstIP d:* If /ii(s, *d)_> c\/{tN)* then ignore the packet. Otherwise:

1. Check entry $\frac{c_1}{t} \cdot h_1(s, d)$ of T_1, and insert (s, d) into the list for this entry if it is not present. Otherwise, it is a duplicate pair and we ignore the packet.
2. At this point we know that d is a new destination for s, i.e., this is the first time (s, d) has appeared in the interval. We use $\frac{c_1}{t} \cdot h_2(s)$ to look-up s in T_2. If s is not found, insert $(s, 1)$ into the list for this entry, as this is the first destination for s in the sample. On the other hand, if s is found, then we increment its count, i.e., we replace the pair (s, k) with $(s, k + 1)$. If the new count equals $r + 1$, we report s. In this way, each declared t-superspreader is reported exactly once.

Note that at the end of the interval, the counts in T_2 can be used to provide a good estimate on the number of distinct dstIPs for each reported srcIP (by scaling them up by the inverse of the sampling rate, i.e., by a factor of tN/c_1).

3.1.2 Analysis

Accuracy Analysis Our analysis yields the following theorem for precision:

Theorem 3.1. *For any given $b > 1$, positive $\delta < 1$, and t such that $b/N < t < 1$, the above algorithm reports srcIPs such that any t-superspreader is reported with probability at least $1 - \delta$, while a srcIP with at most tN/b distinct destinations is (falsely) reported with probability at most δ.*

Please see Appendix II for a detailed proof.

Overhead Analysis The total space is an expected $O(c_1/t)$ memory words. The choice of c_1 depends on b. By equation 1, we have that $c_1 = O(\ln(1/\delta)(\frac{b}{b-1})^2) = O((1 + \frac{1}{(b-1)^2})\ln(1/\delta))$ for $b \leq 3$. For $3 < b < 2e^2$, b is a constant, so $c_1 = O(\ln(1/\delta))$. For larger b, $c_1 = O(\ln(1/\delta))$. Thus across the entire range for b, we have $c_1 = O((1 + \frac{1}{(b-1)^2})\ln(1/\delta))$. This implies that the total space is an expected $O\left(\frac{\ln(1/\delta)}{t}(1 + \frac{1}{(b-1)^2})\right)$ bits. For the typical case where δ is a constant and $b \geq 2$, the algorithm requires space for only $O(1/t)$ memory words.

As for the per-packet processing time, note that each hash table is expected to hold $O(c_1/t)$ entries throughout the course of the algorithm. Thus each hash table look-up takes constant expected time, and hence each packet is processed in $O(1)$ expected time.

3.2 Algorithm II

We have just presented an algorithm that uses a single sampling rate to detect superspreaders. We now present another algorithm that uses *two* sampling rates, and is more memory-efficient in most cases.

At a high level, the algorithm uses two levels of sampling in the following manner: the first-level sampling effectively decides whether we should keep more information about a particular source, while the second-level sampling effectively keeps a small digest that can then be used to identify superspreaders. The first level has a lower sampling rate than the second level. Thus intuitively, the first level is a coarse filter that filters out sources that contact to a small number of distinct destinations, so that we do not need to allocate any memory space for them. The second level is a more precise filter which uses more memory space, and we only use it for sources that pass the first filter.

Intuitively, the reason why Algorithm II is more space-efficient than Algorithm I is because the first sampling rate of Algorithm II is lower than the sampling rate of Algorithm I. If a source

```
function AlgorithmII(s, d)
    Level2(s,d); Levell(s,d);

function Levell(s, d)
    if(^i(s,d) < 7*1) insert s into B\

function Level2(s, d)
    if (h,2(s,d) > $r_2$) return;
    if (5 £ Bi) return;
    For each level 2 bloom filter $B_2$,i in 0,... , 7
        if $h_{3,i}(s,d) < 1/\gamma$
            insert s into B24;
        Count the number y bloom filters B24 which contain s.
        if y > a;, output 5 as a superspreader.
```

Figure 2: Algorithm II pseudocode where (s, d) represents a source-destination pair,

contacts sufficiently many destinations, it will be sampled in both Algorithm I and Algorithm II. But if a source contacts only a few destinations, the probability that it is sampled in Algorithm II is much lower than the probability that it is sampled in Algorithm I. Thus, Algorithm II stores fewer sources that are not superspreaders. It is therefore more space-efficient when there are many sources that contact only a few distinct destinations.

This type of sampling at multiple levels is a new approach that may be of independent interest.

3.2.1 Algorithm Description

The algorithm takes ri,r2,7,cj as parameters, where *r* and r_2 represent the sampling rate in the first and second level respectively, and a; is a threshold. Given the required values for *t* and 6, the values of ri,r2,7,o; will be determined as in Theorem 3.2.

We keep one list *B* at the first level, and 7 lists denoted i?2,i at the second level. Let *h*, ^2^3,15^3,2? ••• 7^3,7 be uniform random hash functions that take a source-destination pair and return a value in [0,1) as described in the previous section.

For each packet (s, d), the network monitor performs the following operations as given in pseudocode in Figure 2:

Step 1: First, we compute ^2(5,*d).* If ^2(5,*d)* is greater than rate *r^* we skip to step 2. Otherwise, we check to see if the source *s* is present in the list *B\.* If s is not present in *B*, then again, we skip to step 2. Otherwise, we insert *s* into level 2 as described below. Thus, we insert *s* into level 2 with at most probability *r*i*, and every source appearing in level 2 appears in level 1.

If /i2(s, *d)* < *T2* and 5 is present in f?i, we consider inserting *s* into each level 2 list i?2,i separately. For each £2,;, we check if *h^i(s,d)* is less than I/7. If *h^i(s,d)* < I/7, we insert *s* into i^,;- Note that, thus, we may insert *s* into multiple level-2 lists. We then count the total number of lists i?2,i that contains *s.* If the number is greater than a;, we output *s* as a superspreader.

Step 2: If /ii(s,d) is less than rate ri, we insert *s* into *B\.*

3.2.2 Optimizations

Note that in the above description, we use lists to store the sampled elements for ease of explanation. We can easily optimize the storage space in the two-level sampling further by using bloom filters instead of lists to store the sampled elements. A discussion of bloom filters may be found in [5]. In addition, in the above description, we chose the probability of inserting a sampled packet into any level-2 list i?2^ to be equal to I/7, for simpler description and analysis. We can easily generalize this to alter the probability of inserting a sampled packet into any level-2 list to be non-uniform, e.g., an exponential distribution.

Another optimization is to reduce the number of times Algorithm **II** must hash the (5, *d)* pair, and thus increase its computational efficiency. Instead of hashing once for each list (or bloom filter) at the second level, we could just directly use the hash value /i$_2$(s, *d)* to choose a single list (or bloom filter) for insertion. The parameters for this optimized algorithm may be chosen in a manner similar to Algorithm II, with similar theoretical bounds. The memory usage of this optimized algorithm is similar to Algorithm **II** as well.

3.2.3 Analysis

Our analysis yields the following theorem for precision:

Theorem 3.2. *Given t such that* 0 < *t* < *1,* TV, *b* > *1, and positive 5* < 1. *Let z* = $\max(\frac{2b}{b-1}, 5)$. *Let* ri = ^ log |, *and* r$_2$ *be minimal value that satisfies the following constraints:*

$$r_2 \geq \frac{2 \text{ In } 2/5}{tN(l \quad - \quad e\text{-}i^{*\text{-}1})/{*}\wedge}$$

$$V2 \; \overset{>}{\sim} \; \frac{\text{In } 1/5}{tN(l \text{ - } e{\sim}^{3}/^{2b})((l + e_2) \ln(l + e_2) \text{ - } c_2)'}$$

$$\text{where } €1 = 1 \frac{1 \;\_\; {}_{e}\text{-}3/26}{1 — 1/e} (1 + e_2),$$

$$e_2 > 0, \textit{ and } 0 < ci < 1.$$

Let e'i be the value of e\ when r$_2$ *is minimized in the above constraints. Let* 7 = *r%tN, and* $\omega = (1 - \epsilon_1')\gamma(1 - \frac{1}{e})$.

Thus, for any given b > *1, positive 5* < *1, and t such that b/N* < *t* < *1, Algorithm II reports srcIPs such that any t-super spreader is reported with probability at least 1—5, while a srcIP with at most tN/b distinct destinations is (falsely) reported with probability at most 5.*

Please see Appendix **III** for a detailed proof.

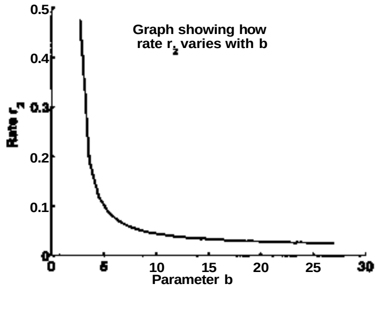


Figure 3: The rate r$_2$ required for *tN* = 1000, *5* = 0.01, with varying 6,

Figure 3 shows how the required rate r$_2$ varies with *b.* The threshold *CJ* and the number of lists 7 vary similarly.

Overhead Analysis The expected space required is *O(riN* + r$_2$iV). Note that, for a fixed b, both 7*1 and r$_2$ are O (^ In $\frac{1}{y}$), and thus the space required is *O{* In |) .

Stream 1: $(s1, d1), (s2, d2), (s3, d3), (s4, d4)$
Stream 2: $(s2, d3), (s1, d1), (s1, d1), (s3, d2)$
Stream 3: $(s4, d2), (s4, d4), (s2, d4), (s4, d3)$

Figure 4: Example streams at 3 monitoring points

We may make a similar statement when we use bloom filters rather than exact lists to store sampled elements as described in the optimization above. Using bloom filters does not affect the false negative rate, but only the false positive rate. We can easily reduce the additional false positive rate caused by the bloom filter collision by setting the correct parameters of the bloom filters using the theorems in [5].

We observe also that the performance guarantees of both algorithms is independent of the input distribution of source-destinations pairs, as long as the assumption of uniform random hash function is obeyed. In addition, note that it is important to pick secret hash functions at run-time each time so that the attacker cannot generate an input sequence that avoid certain hash values.

4 Extensions

In this section we show how to extend our algorithms to handle distributed monitoring, deletions, and sliding windows.

4.1 Distributed Superspreaders

In the distributed setting, we would like to identify source IP addresses that contact a large number of unique hosts in the union of the streams monitored by a set of distributed monitoring points. Consider for example, the three streams in Figure 4 and $t = \frac{1}{5}$. Sources $s1$, $s2$, $s3$, and $s4$ contact 1, 3, 2, and 3 distinct destinations, respectively. Thus for the total of $N = 12$ source-destination pairs, only $s2$ and $s4$ are t-superspreaders.

Note that a source IP address may contact a large number of hosts overall, but only a small number of hosts in any one stream. Source $s2$ in Figure 4 is an example of this. A key challenge is to enable this distributed identification while having only limited communication between the monitoring points.

The following algorithm identifies t-superspreaders in the union of j streams using Algorithm I:

1. Each network monitor runs Algorithm I as described in Section 3.1 on an interval of N/j packets, all using the same hash function h_1. A packet is ignored if $h_1(s, d) \geq \frac{c_1}{tN}$, as in the single stream case. However, because each monitor only sees N/j packets, it expects to store at most $\frac{c_1}{tj}$ entries, so it suffices to have hash tables T_1 and T_2 have only $\frac{c_1}{tj}$ entries. Any locally detected superspreader can be reported (e.g., source $s4$ in stream 3 in Figure 4).

2. At the end of the interval, each monitor sends its hash table T_1 to the centralized point. For $1 \leq i \leq j$, let $T_{1,i}$ be the hash table for stream i.

3. The centralized point merges the information sent by all the monitors and reports t-superspreaders, as follows. It scans through $T_{1,1}, \ldots, T_{1,j}$ and treats the collection of (s, d) pairs as a stream of packets. It performs Algorithm I on this stream (using hash tables of size $\frac{c_1}{t}$), except that it can skip the test of whether $h_1(s, d) \geq \frac{c_1}{tN}$ because the $T_{1,i}$'s only contain pairs that fail this test.

Note that an alternative approach of simply summing the source counts in the hash tables *T2* at each monitor would not be correct, because duplicates would be multiply counted (e.g., in Figure 4, although source si has one distinct destination in stream 1 and one in stream 2, it has only one distinct destination overall, not two).

The overall space and time overhead of step 1 above summed over all the monitors is the same as if one monitor monitors the union of the streams. Step 2 requires a total amount of communication equal to the sum of the space for the Ti/s, i.e., an expected *O{c\/t)* memory words. Accounting for Step 3 increases the total space and time by at most a factor of 2. Note that the algorithm does not require that all streams use an interval of *N/j* packets. As long as there are exactly iV packets in all, the algorithm achieves the precision bounds given in Theorem 3.1. Thus our distributed algorithm uses little memory and little communication.

Algorithm II can be extended similarly.

4.2 Superspreaders with Deletion

We can also extend our algorithms to support streams that include both newly arriving (srcIP, dstIP) pairs and the *deletion* of earlier (srcIP, dstIP) pairs. Recall from Section 2.1 that a motivating scenario for supporting such deletions is finding source IP addresses that contact a high number of distinct destinations and do not get legitimate replies from a high number of these destinations. Each in-bound legitimate reply packet with source IP *x* and destination IP *y* is viewed as a deletion of an earlier request packet with source IP *y* and destination IP *x* from the corresponding flow, so that the source *y* is charged for only distinct destinations without legitimate replies.

For Algorithm I of Section 3.1, a deletion of *(s,d)* is handled by first checking to see if (s, d) is in the list for entry ^ • *hi(s,d)* of *T*. If it is not, then *d* is already not being accounted for in s's distinct destination count, so we can ignore the deletion. Otherwise, we delete (5,0!) from *T*. We use *ϒ ' ^(s)* to look-up *s* in T2, and decrement its count. If its count drops to 0, we remove (5,0) from T2. The precision bounds, space bounds, and time bounds are the same as in the case without deletions.

Similarly, we can handle deletions in Algorithm II of Section 3.2 by deleting from the sample, for the variant where we use lists to store sampled elements.

Note that the definition of a t-superspreader under deletions is not a stable one. At any point in time, the monitor may have just processed a packet, and have no idea whether this pair will be subsequently deleted. There may be a source right at the t-superspreader threshold that exceeds the threshold unless the pair is subsequently deleted. Our algorithms can be readily adapted to handle a variety of ways of treating this issue. For example, Algorithm I can report a source as a *tentative* t-superspreader when its count in *T2* reaches *r* + 1, and then report at the end of the interval which sources (still) have counts greater than r.

4.3 Superspreaders over Sliding Windows

In this section, we show how to extend our algorithms to handle sliding windows. Our goal is to identify t-superspreaders with respect to the most recent *W* packets, i.e., hosts which contact more than *t* • *W* unique destinations in the last *W* packets. Our goal is to use far less space than the space needed to hold all the pairs in the current window.

What makes the sliding window setting more difficult than the standard setting is that a packet is dropped from the window at each step, but we do not have the space to hold on to the packet until it is time to drop it. This is in contrast to the deletions setting described in Section 4.2 where we are given at time of deletion the source-destination pair to delete. In the sliding window

(s1, d1), (sl,d2), (s2,d2), (sl,d3)
(sl,d2), (s2,d2), (sl,d3), (s2,d3)
(s2,d2), (sl,d3), (s2,d3), (s2,dl)

Figure 5: Example stream, showing three steps of a sliding window of size W = 4. The top row shows the packets in the sliding window after the arrival of (sl,d3). The middle row shows that on the arrival of (s2, d3), the window includes this pair but drops (si, dl). The bottom row shows that on the arrival of (s2, dl), the window adds this pair but drops (si, d2).

setting, a source may transition between being a t-superspreader and not, as the window slides. In Figure 5, for example, suppose that the threshold for being a t-superspreader is having at least 3 distinct destinations (e.g., t = .7). Then source si is a superspreader in the first window, but not the second or third windows.

We show how to adapt Algorithm I to handle sliding windows. We keep a running counter of packets that is used to associate each packet with its stream sequence number (seqNum). Thus if the counter is currently x, the sliding window contains packets with sequence numbers x—W+1,... ,#. At a high-level, the algorithm works by (1) maintaining the pairs in our sample sorted by sequence number, in order to find in 0(1) time sample points that drop out of the sliding window, and (2) keeping track of the largest sequence number for each pair in our sample, in order to determine in 0(1) time whether there is at least one occurrence of the pair still in the window. In further detail, the steps of the algorithm are as follows.

- *Initially:* Let L be an initially empty linked list of (srcIP, dstIP, seqNum) triples, sorted by increasing seqNum. Let T\ and *T<±* be as in the original Algorithm I, except that *T* now contains (srcIP, dstIP, seqNum) triples.
- *On arrival of a packet with srcIP s and dstIP d:* Let x be its assigned sequence number.
  1. *Account for a pair dropping out of the window, if any:* If the tail of L is a triple (s, *d,n)* such that n = x — W, then remove the triple from L and check to see if the triple exists in entry ^ • hi(s, *d)* of *T*. If the triple exists, then because *T* holds the latest sequence numbers for each source-destination pair in the sample, we know that (s, *d)* will not exist in the window after dropping (s,d, n). Accordingly, we perform the following steps:
     (a) Remove the triple from *T*.
     (b) Use ^ • *h,2{s)* to look-up s in T2, and decrement the count of this entry in T2, i.e., replace the pair (s, *k)* with (s, k — 1).
     (c) If the new count equals 0, we know that the source no longer appears in the sample and we remove the pair from T2.

     On the other hand, if the triple does not exist, then there is some other triple *(s,d,n^f)* corresponding to a more recent occurrence of (s, *d)* in the stream (n < n'). Thus dropping (s, d, n) changes neither the sampled pairs nor the source counts, so we simply proceed to the next step.
  2. *Account for the new pair being included in the window:* If *h\(s$_1$d)_> c\/{tW)* ignore the packet. Else:
     (a) Check entry ^ • *hi(s,d)* of *T* for a triple with s and d. If such an entry exists, replace it with (s,d,a;), maintaining the invariant that the entry has the latest sequence number, and return to process the next packet. Otherwise, insert (s, d, *x)* into the list for this entry.

(b) At this point we know that d is a new destination for 5, i.e., this is the first time *(sjd)* has appeared in the window. We use ^ • *h,2(s)* to look-up s in *T<i.* If s is not found, insert (s,1) into the list for this entry, as this is the first destination for s in the sample. On the other hand, if s is found, then we increment its count, i.e., we replace the pair (s, *k)* with (s, k + 1). If the new count equals r + 1, we report *s.*

The precision, time and space bounds are the same as in Algorithm I of Section 3.1 with W substituted for *N.*

Note that the algorithm is readily modified to handle sliding windows based on time, e.g., over the last 60 minutes, by using timestamps instead of sequence numbers. The precision, time and space bounds are unchanged, except that the time is now an amortized time bound instead of an expected one. This is because multiple pairs can drop out of the window during the time between consecutive arrivals of new pairs. If more than a constant number of pairs drop out, then the algorithm requires more than a constant amount of time to process them. However, each arriving pair can drop out only once, so the amortized per-arrival cost is constant.

5 Experimental Results

We implemented our algorithms for finding superspreaders, and we evaluated them on network traces taken from the NLANR archive [28], after they were injected with appropriate superspreaders as needed. All of our experiments were run on an Intel Pentium IV, 1.8Ghz. We use the *OPENSSL* implementation of the *SHAl* hash function, picking a random key during each run, so that the attacker cannot predict the hashing values. For a real implementation, one can use a more efficient hash function. We ran our experiments on several traces and obtain similar results. Our results show that our algorithms are fast, have high precision, and use a small amount of memory. On average, the algorithms take on the order of a few seconds for a hundred thousand to a million packets (with a non-optimized implementation).

In this section, we first examine the precision of the algorithms experimentally, then examine the memory used as *k, b* and N change, and finally compare with the alternate approach proposed in Section 2.3.

5.1 Experimental evaluation of precision

To illustrate the precision of the algorithms, we show a set of experimental results below. To the base trace 1 (see Table 3), we inserted various attack packets where some sources contacted a high number of distinct destinations. That is, for given parameters k = *t-N* and 6, we added 100 sources that sends to k destinations each, and 100 sources that send to about *k/b* destinations each. This was done in order to test if our algorithms do indeed distinguish between sources that send to more than k destinations and fewer than *k/b* destinations.

We set *6* = 0.05. In the Table 2, we show the results of our experiments, with regards to precision of the algorithms. We examine the correctness of our algorithm by comparing it against an exact calculation. The other parameters are set as given by Theorems 3.1 and 3.2.

We observe that the accuracy of both algorithms is comparable and bounded by 5, which confirms our theoretical results. We note that the false positive rate is much lower than the false negative rate. Our sampling rates are chosen to distinguish sources that send to k destinations from sources that send to *k/b* destinations with error rate *5.* When a source sends to a very small number of destinations (much smaller than fc/6), the probability that it becomes a false positive is significantly lower than *8.* Likewise, when a source sends to a very large number of

k = *t*-*N*	*b*	False Positives		False Negatives	
		Algl	**Algll**	Algl	Algll
500	*2*	**8.1e-4**	6.3e-5	0	0
500	**5**	2.75e-4	2.43e-4	0	0
500	10	1.35e-4	1.94e-4	0.01	0
1000	2	4.95e-5	1.13e-4	0.02	0
1000	5	1.62e-4	0	0.02	0.02
1000	10	4.95e-4	9.45e-5	0	0.03
5000	2	6.75e-4	0	0	0
5000	5	4.95e-5	1.62e-5	0	0
5000	10	3.19e-5	1.62e-5	0	0.01
10000	2	1.62e-4	0	0.02	0
10000	5	3.2e-5	0	0.01	0
10000	10	**1.62e-5**	3.2e-5	0.04	0

Table 2: Evaluation of the Precision of Algorithms I and II over various settings for *t* and *b*, with *5* = 0.05.

	No. distinct sources	No. distinct src-dst pairs	*N* (no. of packets)
Trace 1	59,862	194,060	2.88 x 10^b
Trace 2	282,484	416,730	3.09 x 10^b
Trace 3	1.21 x 10^b	1.35 x 10^6	4.02 x 10^6
Trace 4	2.12 x 10^6	2.29 x 10^6	4.49 x 10^6

Table 3: Base traces used for experiments

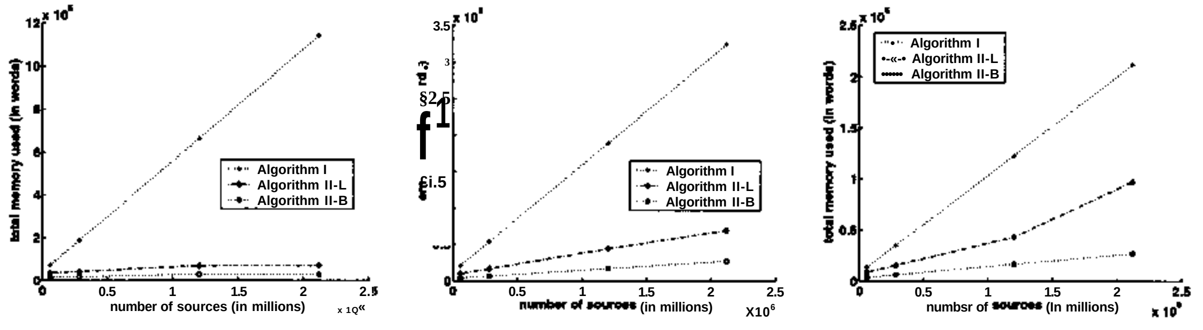
destinations (^> fc), the probability that it becomes a false negative is much less than *S*. Through the construction of our traces, there are only a 100 possible sources that may be false negatives, and all of them send to just over *k* destinations. There are many more sources that could be false positives, and only a 100 of these sources send to nearly *k/b* destinations. Thus, the false positive rate that is seen is much less than the set *5*,

5.2 Memory **usage on long traces**

We now examine memory used on very long traces by Algorithm I (Section 3.1) and the list and bloom-filter implementations of Algorithm II (Section 3.2). To distinguish the two implementations of Algorithm II, we will refer to the list implementation as Algorithm II-L, and the bloom-filter implementation as Algorithm II-B. We examine the memory used as the parameters fc, *b* and JV axe allowed to vary.

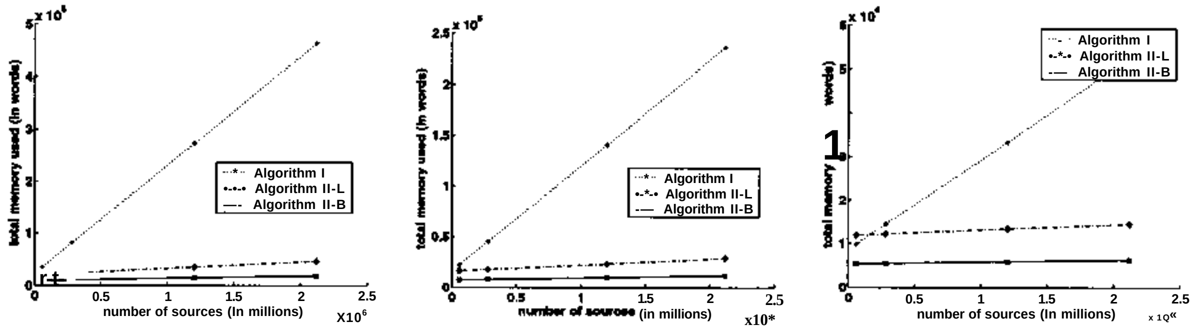
The traces used for this section are constructed by taking four base traces of varying lengths, and adding to each of them a hundred sources that send to *k* destinations, and a hundred sources that send to *k/b* destinations. The details of the base traces are shown in Table 3. We observe that, with the largest of these traces, a source that sends to 200 distinct destinations contributes just about 0.004% to the total traffic analyzed. The memory used is the number of words (or IP addresses) that need to be stored.

The graphs in Figure 6 show the total memory used by each algorithm plotted against the number of distinct sources in the trace, at different values of *b*. Notice that through our trace construction procedure, the traces in Figure 6(a), 6(b), and 6(c) contain the same number of



(a) 6 = 2 (b) b = 5 (c) b = 10

Figure 6: Total memory used by the algorithms in words (i.e., IP addresses) vs number of distinct sources, for b = 2, 5 and 10, at k = 200.



(a) k = 500 (b) k = 1000 (c) k = 5000

Figure 7: Total memory used by the algorithms in words (i.e., IP addresses) vs number of distinct sources, for k = 500,1000 and 5000, at b = 2.

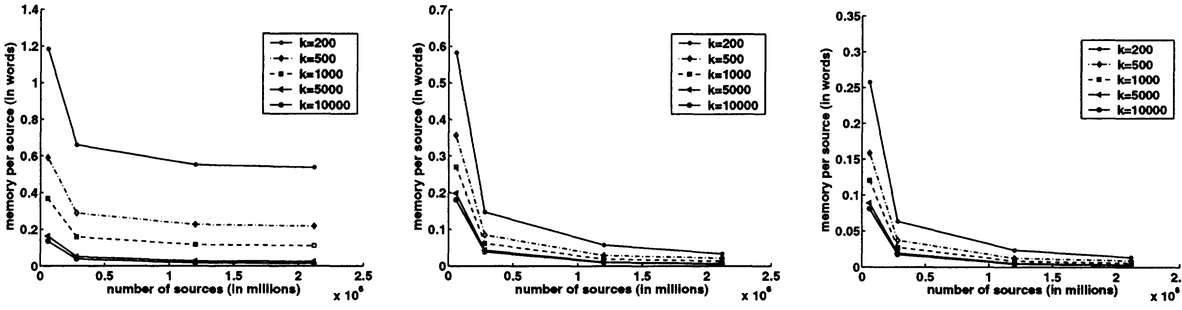
distinct sources, even though the value of b differs.

We observe that the memory used by the two algorithms is strongly correlated with 6, as pointed out by our theoretical analysis. For both algorithms, the memory required decreases sharply as b increases from 2 to 5, and then decreases more slowly. This can also be seen (for Algorithm II) from Figure 3, in section 3.2.

Another observation is that, as expected, the memory used by Algorithm I eventually exceeds the memory used by Algorithms II-L & II-B, for every value of b. The number of sources at which the memory used by Algorithm I exceeds the memory used by Algorithms II-L & II-B also depends on b. We also note that, as expected, the memory used by Algorithm II-B is much less than the memory used by Algorithm II-L and Algorithm I.

We next examine the memory usage as k changes, which is shown in Figures 7 and 8. We observe that the total memory used drops sharply as k increases, as expected: in 7(a), at k = 500, the memory used ranges from 20,000 to 200,000 IP addresses; in 7(c), at k = 5000, it ranges from 10,000 to 55,000 IP addresses. Even though the number of source-destination pairs increases when k increases, we can afford to sample much less frequently. This in turn decreases the number of sources stored that have very few destinations, and thus the total memory used decreases.

Also, for every &, as the number of packets N increases, the memory used by Algorithm I eventually exceeds the memory used by Algorithm II-L & II-B. This is because of the two-level sampling scheme. Since the first sampling rate $r\$ is much smaller than ci/k in Algorithm I, the number of non-superspreader sources stored in the Algorithm II (nN in expectation) is much less than in Algorithm I. The actual number of sources at which this occurs depends on k. As k



(a) Alg I

(b) Alg II-L

(c) Alg II-B

Figure 8: Memory used per source vs number of distinct sources, for all k, by Algorithms I, II-L, & II-B at $b = 2$.

b	Alg I	Alg II-L	Alg II-B	Alt I	Alt II
Trace 1					
2	37610	16234	7223	49063	105589
5	9563	3241	1377	20746	48424
10	5685	2698	1136	16839	36823
Trace 2					
2	71852	17536	7711	133988	273101
5	19298	4543	1865	76543	168256
10	12030	4000	1624	67007	135540

Table 4: Comparisons of total memory used with traces 1 & 2 for $k = 1000$ and varying b.

increases, the number of sources at which the memory used by Algorithm I exceeds the memory used by Algorithm II also increases, since the sampling rates for both algorithms decrease in the same way. We also observe that, once again, the memory used by Algorithms II-B is significantly lower than Algorithm I and Algorithm II-L.

The graphs in Figure 8 show the memory used per source plotted against the number of distinct sources, for various k – as k increases, the total memory used drops. We observe that each algorithm has a similar dependence on k, though the absolute memory usage is different, as discussed.

5.3 Comparison with an Alternate Approach

We now show some results comparing our approach to the approach described in Section 2.3: we count the number of distinct destinations that a source sends to using LossyCounting [25], replacing the regular counter with a distinct-element counter. The details are in Appendix IV.

We chose the parameters for LossyCounting and the distinct counting algorithm so that (a) the memory usage was minimized for each b and (b) they produced the false positive rates similar to Algorithms I & II over 10 iterations. We show experimental results with two variants of the approach: (1) use one distinct counter per source (this is Alt I), and (2) use $\log \frac{1}{\delta}$ distinct counters per source, and use their median for the estimate of the number of distinct elements is used (this is Alt II). The memory used is reported as the maximum of the total number of hash values stored for all the sources at any particular time.

Table 4 shows the result of the comparison of memory usage at $k = 1000$, for $b = 2, 5$ and 10, on Trace 1 & Trace 2. Note that all our algorithms show better performance than Alt I and Alt II on Traces 1 & 2. The results for Trace 3 & Trace 4 are similar, except that Alt I uses less memory than Algorithm I when *b* = 2. We explain why Alt I is better than Alt II in Appendix IV.

6 Related Work

We review related work in this section. There has been a volume of work done in the area of streaming algorithms. However, to the best of our knowledge, no previous work directly addresses the problems we address in this paper.

Distinct values counting has been studied by a number of papers *(e.g.,* [1, 3, 4, 8, 10, 17, 19, 20, 16]). The seminal algorithm by Flajolet and Martin [17] and its variant due to Alon, Matias and Szegedy [1] estimate the number of distinct values in a stream (and also the number of l's in a bit stream) up to a relative error of e > 1. Cohen [7], Gibbons and Tirthapura [19], and Bar-Yossef et al. [4] give distinct counting algorithms that work for arbitrary relative error. Of these, the algorithm in [19] has the best space and time bounds: their (e, <5)-approximation scheme uses *O(* log(l/5) logi?) memory bits, where the values are in [0../2], and an expected 0(1) time per stream element. More recently, Bar-Yossef *et al.* [3] improve the space complexity of distinct values counting on a single stream, and Cormode *et al.* [8] show how to compute the number of distinct values in a single stream in the presence of additions *and deletions* of items in the stream. Gibbons and Tirthapura [20] give an (e, (^-approximation scheme for distinct values counting over a *sliding window* of the last *N* items, using *B* = *O(-^\og(l/S)logNlogR)* memory bits. The per-element processing time is dominated by the time for <9(log(l/<5)) finite field operations. The algorithm extends to handle distributed streams, where *B* bits are used for each stream.

There are a number of papers that have proposed algorithms for other problems in network traffic analysis as well. Estan et al. [16] present a series of bitmap algorithms for counting the number of distinct flows for different applications. Estan and Varghese [15] propose two algorithms to identify the large flows in network traffic, and give an accurate estimate of their sizes. Estan et al. [14] present an offline algorithm that computes the multidimensional traffic clusters reflecting network usage patterns. Duffield et al. [12] show that the number and average length of flows may be inferred even when some flows are not sampled, and in [13] they compute the entire distribution of flow lengths. Golab et al. [21] present a deterministic single-pass algorithm to identify frequent items over sliding windows. Cormode and Muthukrishnan [9] present sketch-based techniques to identify significant changes in network traffic.

There have been many papers for other problems involving data streaming algorithms (see the surveys in [2, 27]). None of this work has addressed the problems considered in this paper.

7 Conclusion

In this paper, we have described two new streaming algorithms for identifying superspreaders on high-speed networks. Our algorithms give proven guarantees on the accuracy and the memory requirement. Compared to previous approaches, our algorithms are substantially more efficient, both theoretically and experimentally. The best previous approach [16] requires a certain amount of memory to be allocated for each source within the time window. Our algorithms are the first ones where the amount of memory allocated is independent on the number of sources in the time window. We also provide several extensions to our algorithms - we provide solutions to identify superspreaders in the distributed setting, as well as solutions to allow deletions in the stream and solutions for the sliding window case. We are the first ones to give solutions for these extensions. Our algorithms have many important networking and security applications. We also hope that our algorithms will shed new light on developing new fast streaming algorithms for high-speed networking and security monitoring.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. of Computer and System Sciences,* 58:137-147, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issue in data stream systems. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS),* pages 1-16, June 2002.

[3] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques (RANDOM),* pages 1-10, September 2002. Lecture Notes in Computer Science, vol. 2483, Springer.

[4] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA),* January 2002.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton,* 2002.

[6] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th Intl. Colloq. on Automata, Languages, and Programming,* 2002.

[7] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. of Computer and System Sciences,* 55(3):441-453, 1997.

[8] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *Proc. 28th International Conf. on Very Large Data Bases (VLDB),* pages 335-345, August 2002.

[9] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. In *Proceedings of IEEE Infocom'04,* 2004.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing,* 31(6):1794-1813, 2002.

[11] E. Demaine, A. Lopez-Ortiz, and J. Ian-Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms,* pages 348-360, September 2002.

[12] N. Duffield, C. Lund, and M. Thorup. Properties and prediction of flow statistics from sampled packet streams. In *A CM SIGCOMM Internet Measurement Workshop,* 2002.

[13] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of ACM SIGCOMM,* 2003.

[14] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of SIGCOMM'03,* 2003.

[15] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of SIGCOMM*^{J}*02,* 2002.

[16] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *A CM SIGCOMM Internet Measurement Workshop,* 2003.

[17] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Computer and System Sciences,* 31:182-209, 1985.

[18] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conf. on Management of Data,* pages 331-342, June 1998.

[19] P. B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA),* pages 281-291, June 2001.

[20] P. B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA),* pages 63-72, August 2002.

[21] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. Ian-Munro. Identifying frequent items in sliding windows over online packet streams. In *Proceedings of 2003 ACM SIGCOMM conference on Internet measurement,* pages 173-178. ACM Press, 2003.

[22] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *A CM Trans. Database Syst.,* 28(l):51-55, 2003.

[23] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement,* pages 234-247. ACM Press, 2003.

[24] A. Kumar, L. Li, and J. Wang. Space-code bloom filter for efficient traffic flow measurement. In *Proceedings of IEEE Infocom'04,* 2004.

[25] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of VLDB 2002,* 2002.

[26] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. Security and Privacy Magazine, July/August 2003.

[27] S. Muthukrishnan. Data streams: Algorithms and applications. Technical report, Rutgers University, Piscataway, NJ, 2003.

[28] NLANR. National laboratory for applied network research. http://pma.nlanr.net/Traces/, 2003.

[29] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference.* USENIX, 1999.

A Analysis background

In the analysis of our algorithms, we will use the following Chernoff bounds.

Fact A.1. *Let X be the sum of n independent Bernoulli random variables with success probability p. Then for all $\beta > 1$,*

$$Pr[X \geq \beta np] \quad \leq \quad e^{(1-1/\beta-\ln\beta)\beta np} \tag{3}$$

Moreover, for any ϵ, $0 < \epsilon < 1$,

$$Pr[X \geq (1+\epsilon)pn] \quad \leq \quad e^{-\epsilon^2 np/3} \tag{4}$$

and

$$Pr[X \leq (1-\epsilon)pn] \quad \leq \quad e^{-\epsilon^2 np/2} \tag{5}$$

B Proof for Theorem 3.1

Proof. Each distinct (srcIP, dstIP) pair occuring during the interval is retained according to a Bernoulli trial with success probability $\frac{c_1}{tN}$. The probability of a srcIP being reported increases monotonically with its number of distinct destinations. Thus it suffices to show that

P1. *False negatives:* the probability that a srcIP s_1 with tN distinct destinations has at most r successes is at most δ, and

P2. *False positives:* the probability that a srcIP s_2 with tN/b distinct destinations has at least r successes is at most δ.

We seek to achieve P1 and P2 while keeping c_1 small. Let X_1 be the number of successes for s_1 and let X_2 be the number of successes for s_2. Let $k = tN$. We consider each of the three ranges for b in turn.

Consider the case when $b \leq 3$. By equation 5, we have $Pr[X_1 \leq (1-\epsilon_1)(c_1/k)k] \leq e^{-\epsilon_1^2 k(c_1/k)/2}$ for any ϵ_1 between 0 and 1. Setting $e^{-\epsilon_1^2 c_1/2} = \delta$ and solving for ϵ_1, we get $\epsilon_1 = \sqrt{\frac{2}{c_1}\ln(1/\delta)}$. Because $\delta < 1$, we have that $\ln(1/\delta) > 0$. Thus as long as $c_1 > 2\ln(1/\delta)$, we have $0 < \epsilon_1 < 1$. By equation 4, we have $Pr[X_2 \geq (1+\epsilon_2)(c_1/k)(k/b)] \leq e^{-\epsilon_2^2(k/b)(c_1/k)/3}$ for any ϵ_2 between 0 and 1. Setting $e^{-\epsilon_2^2 c_1/(3b)} = \delta$ and solving for ϵ_2, we get $\epsilon_2 = \sqrt{\frac{3b}{c_1}\ln(1/\delta)}$. As long as $c_1 > 3b\ln(1/\delta)$, we have $0 < \epsilon_2 < 1$.

Because the same cut-off r is used for s_1 and s_2, we require that $r = (1-\epsilon_1)c_1 = (1+\epsilon_2)(c_1/b)$, i.e., that $b\left(1 - \sqrt{\frac{2}{c_1}\ln(1/\delta)}\right) = 1 + \sqrt{\frac{3b}{c_1}\ln(1/\delta)}$. Thus, $b\sqrt{c_1} - b\sqrt{2\ln(1/\delta)} = \sqrt{c_1} + \sqrt{3b\ln(1/\delta)}$. Solving for c_1, we get $\sqrt{c_1} = \sqrt{\ln(1/\delta)}(\sqrt{3b} + b\sqrt{2})/(b-1)$, and hence

$$c_1 = \ln(1/\delta)\left(\frac{3b + 2b\sqrt{6b} + 2b^2}{(b-1)^2}\right) \text{ when } b \leq 3 \tag{6}$$

This is the smallest c_1 that works for both s_1 and s_2, when applying the above Chernoff bounds (equations 4 and 5). Because $b > 1$ and $2b^2/(b-1)^2 > 2$, we have that $c_1 > 2\ln(1/\delta)$. To show that $c_1 > 3b\ln(1/\delta)$ when $b \leq 3$, we must show that $3b + 2b\sqrt{6b} + 2b^2 > 3b(b-1)^2$, i.e., $3 + 2\sqrt{6b} + 2b > 3b^2 - 6b + 3$, i.e., $2\sqrt{6} > (3b-8)\sqrt{b}$. Now when $b \leq 3$, we have $3b - 8 \leq 1 < 2$ and *y/b* < A76, and hence *c* > 361n(l/<5). It follows that for the *C* in equation 6, PI and P2 hold when r = $(1+\epsilon_2)(c_1/b) = \frac{c_1}{b} + \sqrt{\quad}$ ^ln(1/δ).

Next consider the case when 3 < *b* < 2e^2. As argued above, PI holds with *e* = $\sqrt{J2\frac{}{c_1}}$ ln(l/<J), as long as *c* > 21n(l/5). (This analysis did not depend on 6.) On the other hand, note that we cannot use the same analysis as in the previous case to show P2 holds because for example, when 6 = 4, the *c* in equation 6 is less than 36 In(1/5). Instead, we apply equation 3 with *0 = e: Pr[X$_2$ >eci/b] <* $_e$(i-i/g-i)"*/* = *e~^{Cl}/^{b}*. We require that r = *eci/b* = (1 - ci)ci, i.e., 6i = 1 - *e/b* = $\sqrt{}$*j£]n(l/6).* Solving for ci, we get *c* = 21n(l/<J)/(l - f)^2. Moreover, PI holds for aHci>21n(l/5)/(l-f)^2.

Similarly, setting *e~^{Cljfb} = 5* and solving for ci, we get that *c* = *bln(l/S)* and moreover, P2 holds for all *c* > 61n(l/5). Thus selecting *c* such that

$$d \quad = \quad \text{ln(l/5)-max(6,2/(l-e/6)}^2\text{)} \qquad \text{when } 3 < 6 < 2e^2 \qquad (7)$$

implies both PI and P2 hold when *r* = *eci/b.* (Note that 0 < (l-e/6)^2 < 1 and hence *c* > 2 ln(l/5), as required for PI.)

Finally, consider the case when 6 > 2e^2. Note that the *c* from equation 7 grows linearly in 6. Thus for large 6, we seek a modified analysis in which *c* does not grow asymptotically with 6. First we apply equation 3 with *0* = 6/2: *Pr[X$_2$ >* (6/2)(ci/6)] < $_e$(i-2/&-in(6/2))$_{Cl}$/2 < $_e$-$_{Cl}$/2 because 6 > 2e^2 implies ln(6/2) > 2 implies 1 - 2/6 - ln(6/2) < - 1 . Thus *r* = ci/2 = (1 - ei)ci, i.e. ei = 1/2. By equation 5, we have *Pr[X$_1$ < r]* < e-C^1/^{2})^2^/^{2} = e~$^{Cl/8}$. It follows that selecting *d* = 81n(l/(S) when 6 > 2e^2 implies that both PI and P2 hold when *r* = ci/2. D

C Proof for Theorem 3.2

We begin our analysis by making an observation. Our sampling is done by hashing source-destination pairs; therefore, if the same source-destination pair appears multiple times, its chances of being sampled do not change. Thus, hashing effectively reduces all the packets in the stream to a set of distinct source-destination pairs, and in this transformed set, a superspreader appears at least *k* = *tN* times. Effectively, we sample only from this transformed set. Therefore, we analyze the algorithm in this transformed set, in which all source-destination pairs are distinct.

Proof. We consider Algorithm II in Section 3.2, under the parameters given by the theorem. We will first analyze the false negatives, and then the false positives.

False negatives. We analyze the false negative error in two parts. For a source *i* with *ni* > *k*, we first show that the source is inserted into Li, with probability 1 - |, in the first *1/z* fraction of its total pairs. Then, we show that it is inserted into at least *u* of the lists *L$_2$^i* over the rest of its pairs, with probability 1 — |, conditioned on its presence in *L*. Together, these two parts ensure that, with probability at least 1 — <5, the source is present in *UJ* of the lists *L$_2$,i,* after all the distinct destinations for that source axe seen.

For the first part of the false negative analysis, we need to set the rate of sampling ri so that any source that appears more than *k* times in the sampled set will be sampled with a probability of 1 — |, within its first \ fraction of packets. This is equivalent to saying that any source *s* with | packets is present in Li with probability at least 1 - |. Equivalently, 1 — *Pr[s* G *Li\n$_8$ >_k/z]* = (1 — *ri)$^{\frac{k}{z}}$*

which needs to be bounded by |. Therefore,

$$r_1 \geq \frac{z}{k} \log \frac{2}{\delta}.$$

This gives us a lower bound on n. We want *r* to be as small as possible to minimize the memory needed, so we set *r* to its lower bound.

Now we analyze the second part of the false negative error. By setting *r* as specified, we know that the source will be present in *L* with probability 1 - | in the first $\frac{k}{z}$ packets. So, in the remaining fc(l — *)* packets, we examine the probability that the source will fall into *u* lists in the second step, conditioned on the event that the source is already present in *L*.

Let *Xij* be the indicator variable for the event that source *i* is put into the jth list, *L2*j, when the source is already present in list *L*. The expected number of lists L2J that will contain source *i* is the sum *Sj* = *Yll=i Xij-*

Let *k!* = fe(l - *\).* For a source *i* with *m* ≥ *k': Pr[Xij* = 1] = 1 - *Pr[Xij* = 0]_> 1 - (1 - ^$)^{k'}$ which is at least 1 — $e\sim^{V2k'}$∧.

Therefore, for a source *i* with *rti* ≥ fc', we can write the expected value of lists set *E[Si* = *Yl]=i*^{E}*[*^{x}*ij}* >_7C^{1} - e$^{"r2fc/}$/7). Let /xi = *E[Si].* The random variable *Si* is equivalently the result of sampling 7 coins of bias *Pr[Xij* = 1], so we can use the Chernoff bounds to get a lower bound on *Si,* with with probability at least 1 — |.That is, for *e* £ (0,1), by Chernoff bounds: $Pr[\ S \leq \{1 + \epsilon_1\} \ \leq e^{\frac{-\mu_1\epsilon_1^2}{2}}$.

Let CJI = (1 — *e\)fJL*. We need to have *Pr[Si_<* (1 — €1)^1]_< §. We can get this by setting $e^{\frac{-\mu_1\epsilon_1^2}{2}}$ ≤ |. Therefore, *ji* ≥ $\frac{\backslash}{\epsilon_1}$log |. Since *ji* >_7(1 — $e^{\frac{-r_2k'}{1}}$) we need to have 7(1 — $e\sim^{r2k'}$∧) ≥ $\frac{\backslash}{\epsilon_1}$log I. Since 7 = r2fc, *r2k{* — $e\sim^{k'}$/^{k}*)*_> $\frac{\backslash}{\epsilon_1}$ log |. Therefore,

2 2
ke{(l — $e\sim^{k}$ *f*^{k}*)* *0*

Thus, we get the first constraint in the optimization problem in Theorem 3.2.

False positives. Now we analyze the false positive error. A source *i* is a false positive if *Ui* < *k/b,* but it is still identified as a superspreader by our algorithm. To bound this error, it is enough to show that the source is present in *u*> of the lists *Lij,* with sufficiently low probability. Therefore, we compute a constraint on the rate of sampling *r*^ so that, with probability at least 1 — 5, no more than *u)* lists will contain the source. We note that the probability of being a false positive is maximized when *n{* is as large as possible, so we assume now that *ni* = *k/b.*

As in our analysis for the false negatives, let Xy be the indicator variable for the event that the source *i* is put into the *jth* list in L2: *X*^ is 1 if the source is put into the jth list, and 0 otherwise. Once again, we can compute the probability that source *i* gets put into the jth list L2J as follows: *Pr[Xij* = 1] = 1 - *Pr[Xij* = 0]_< 1 - e$^{"3r2fc}$/^{2}^.

Therefore, the expected number of lists containing source *i* with n* = | can be written as *E[Si]* = *E]=iE[Xij]* ≤ 7(1 - $_{e}$-^{3r2fc}/^{2}^). Substituting for 7, we get *E[Si]* < *r*$_{2}$*k(l* - $\overset{-1}{e}$^) = $r_2k(1 - e^{-3/2b})$.

Let *H2* — *E[Si*. Once again, with Chernoff bounds, we can get, for *€2* > 0, *Pr[S{ >_(1 + 62)1^2]* ^ e$^{"c}$∧^{2}, where c = -loge^{e2}(l + e$_{2}$)$^{"(1+e2)}$

Let u>2 = (1 + €2)^2. Since we want to bound *Pr[Si* ≥ (1 + ^2)^2] by |, this becomes *S* ≥ $_{e}$-$_{M}$2((i+e2)iog(i+e$_{2}$)-e$_{2}$)$_{>}$ Substituting for /x$_{2}$, we get log £_< r$_{2}$fc(l - e~^{3}/^{26})((l + c$_{2}$) log(l + c$_{2}$) - c$_{2}$).

Thus,

$$r_2 \geq \frac{1}{k(1-e^{-3/2b})} \frac{1}{((1+\epsilon_2)\log(1+\epsilon_2)-\epsilon_2)} \log\frac{1}{\delta}.$$

Thus, we have the second constraint for r_2.

We finally have to establish the relationship between ϵ_1 and ϵ_2. This we get by equating the definitions of ω_1 and ω_2, since the algorithm uses only one threshold ω:

$$\begin{aligned} \mu_1(1-\epsilon_1) &= \mu_2(1+\epsilon_2) \\ (1-e^{-1})(1-\epsilon_1) &= (1-e^{-\frac{3}{2b}})(1+\epsilon_2) \\ \epsilon_1 &= 1-\frac{1-e^{-3/2b}}{1-e^{-1}}(1+\epsilon_2). \end{aligned}$$

Thus, we get the last relation in the problem.

We wish to minimize r_2, subject to these constraints, because the expected memory is $O(r_2N)$. Therefore, the solution to the problem in Theorem 3.2, gives us a sampling rate r_2 such that a source with k distinct destinations will be in at least ω lists, and a source sending to less than k distinct destinations will not be in ω lists, with probability at least $1-\delta$. □

D LossyCounting and Distinct Counting Algorithm

The following is a summary of LossyCounting and the distinct counting algorithms that we use in our experimental comparisons in Section 5.3.

- *LossyCounting*: The stream of elements is divided into *epochs*, where each epoch contains $\frac{1}{\epsilon}$ elements; thus, for an input stream of N elements, we will have ϵN epochs. Each epoch has two phases: in the first phase, the incoming elements are simply stored, and if already present, their frequency is updated; in the second phase, the algorithm looks over all elements, and discards those with a low frequency count. It can be shown that the final frequency count of the elements is at most ϵ lower than the true frequency count, and clearly, it cannot be larger than the true frequency count.

- *Distinct Counting*: Every element in the input stream is hashed (uniformly) between $(0,1)$, and the t lowest hash values are stored. At the end, the algorithm reports t/min_t as the number of distinct elements in the stream, where min_t is the value of the tth smallest hash value. In order to get an (ϵ,δ)-approximate answer, the authors show that t needs to be no larger than $96/\epsilon^2$, when $O(\log\frac{1}{\delta})$ copies of the algorithm are run in parallel.

- *Putting them together*: In order to find superspreaders using LossyCounting, we need to replace the regular frequency counter in LossyCounting with a distinct counter. Therefore, when a source-destination pair is examined, the source and destination is hashed, and the t smallest hash values (of any particular source) are stored. At the end of each epoch, all the sources whose counts of distinct destinations are below the threshold set by LossyCounting are discarded. We need to run $O(\log\frac{1}{\delta})$ copies of the distinct counter per source, and use the median value of the multiple copies. At the end, all sources whose threshold exceeds $k/b+\epsilon$ are returned, where ϵ comes from the error in distinct-counting. The tolerable error in LossyCounting determines the number of epochs (and therefore, space required), and we set these error parameters so that the expected memory usage is minimized.

We show experimental results with two algorithms based on this approach: (1) use one distinct counter per source (Alt I), and (2) use log ^ distinct counters per source, and use their median for counting approximately the number of distinct elements (Alt II). The distinct counting algorithm requires O(log |) parallel runs for its guarantees. However, Alt I is always better than Alt II in our experiments. This is because many sources send packets to only a few destinations, and for those sources, there is log ^ factor increase in the memory usage, even though the actual constant *t* decreases.