New Streaming Algorithms for Fast Detection of Superspreaders

Shobha Venkataraman*, Dawn Song*, Phillip B. Gibbons†, Avrim Blum*
*Carnegie Mellon University
{shobha@cs, dawnsong@ece, avrim@cs}.cmu.edu
†Intel Research Pittsburgh
phillip.b.gibbons@intel.com

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 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 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 extend our algorithms to identify superspreaders in a distributed setting, with sliding windows, and when deletions are allowed in the stream (which lets us identify sources that make a large number of failed connections to distinct destinations).*

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 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 against 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, in order 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, in order to throttle subsequent (similar) attack traffic in real-time.

In this paper, we study the problem of identifying *superspreaders*. A superspreader is defined to be a host that contacts at least a given number of distinct destinations within a short time period. Superspreaders could be responsible for fast 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.

Note that a superspreader is different from the usual definition of a heavy-hitter ([19, 8, 16, 25, 13, 24]). 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 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

$(s1, d1), (s2, d2), (s1, d1), (s3, d3), (s1, d1), (s2, d3), (s4, d1), (s2, d4), (s1, d1), (s5, d4), (s6, d6)$

Figure 1. Example stream of (source, destination) pairs, starting with $(s1, d1)$ and ending with $(s6, d6)$.

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.

It is desirable to be able to do the monitoring on high-speed links, for example, on a large enterprise network or an ISP network for a large number of home users. A major difficulty with detecting superspreaders on a high-speed monitoring point is that 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. The attack traffic may be an extremely small portion of the total traffic. 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. We need, therefore, efficient algorithms to find superspreaders that use memory sparingly.

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.

An algorithm for the heavy distinct-hitters problem has a wide range of applications. Clearly, we can solve the dual of the superspreader problem – finding the destinations which are contacted by a large number of sources – and such destinations could be victims of DDoS attacks. 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). Such an algorithm 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. An algorithm for the heavy distinct-hitter problem may also be useful in peer-to-peer networks, where it could be used to find nodes that talk to a lot of other nodes without keeping per-node information. For simplicity, in the rest of this paper, we will describe our algorithms for identifying superspreaders. The algorithms can be easily applied to the other applications mentioned above.

To summarize, the contributions of this paper are the following:

- We propose new streaming algorithms for identifying *superspreaders*. Our algorithms are the first to address this problem efficiently and provide proven accuracy and performance bounds. The best previous approaches [17, 31] require a certain amount of memory to be allocated for each source [17] or each flow [31] within the time window; we do not keep state for every source, and thus our algorithms scale very well. We present two algorithms: the first, a simpler one, which is already much better than existing approaches, and which we use for base comparison; and the second, a more complex two-level filtering scheme, that is more space-efficient on commonly-seen distributions. In addition, the two-level filtering scheme may have other applications and be of independent interest.

- We also propose several extensions to enhance our algorithms – we extend our algorithms to scenarios when deletion is allowed in the stream (Section 4.1), to the sliding window scenario (Section 4.2), and we propose efficient distributed versions of our algorithms (Section 4.3). The deletion scenario is especially well-motivated – it can be used to find sources that have a large number of distinct connection failures (this may be an indication of scanning behavior), rather than just sources that contact a large number of distinct destinations. That is, once the network monitoring point sees a response from a destination for a connection from a source, that source-destination pair gets deleted from the count of the number of distinct connections a source makes.

- Our experimental results on traces with up to 10 million flows confirm our theoretical results. Further, they show that the memory usage of our algorithms is substantially smaller than alternative approaches. Finally, we 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 attack detection and defense,

when *given* certain parameters. Selecting and testing the correct parameters, however, is application-dependent and outside of the scope of this paper.

Note that we cannot detect a malicious host that spoofs IP addresses and contacts many destinations, since the algorithms will only operate on the input (src, dst) pairs. It is, however, difficult to engage in TCP-based attacks with IP spoofing. Also, we may need special care when identifying the connection direction. We can handle this issue in TCP traffic by checking for superspreaders only in the SYN packets. In UDP traffic, though, this may not be possible, because we may not be able to distinguish which of the two hosts sent the first packet without extra storage. Thus, in UDP traffic, we may not be able to distinguish between a superspreader and a source that simply *responds* to many clients. (In our abstraction, the latter is also a superspreader in case of UDP). In practice, though, we expect that most sources that typically need to respond to many clients will remain more or less constant over brief periods of time (e.g. web servers over a few days' time), and that it will be easy to identify these sources early, and keep them on a separate list, so that they do not interfere in network anomaly detection.

The rest of the paper is organized as follows. Section 2 defines the superspreader problem and discusses previous approaches. 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, and Section 6 presents conclusions.

2 Problem Definition and Previous Approaches

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

2.1 Problem Definition

We define a *k-superspreader* as a host which contacts more than k unique destinations within a given window of N source-destination pairs. In Figure 1, for example, with $k = 2$, source $s2$ is the only k-superspreader. Note that there may be as many as N/k k-superspreaders in a given set of N packets, and reporting them would need $\Omega(N/k)$ space. Thus, this gives us a lower bound on the space bounds needed to find superspreaders. It also 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 $\Omega(k)$ space. Because we are interested in small space algorithms, we must consider instead randomized algorithms.

N	Total no. of packets in a given time interval
k	A superspreader sends to more than k distinct destinations
b	A false positive is a source that contacts less than k/b distinct destinations but is reported as a superspreader
δ	Probability that a given source becomes a false negative or a false positive
W	Sliding window size
s	Source IP address
d	Destination IP address

Table 1. Summary of notation

More formally, given a user-specified $b > 1$ and confidence level $0 < \delta < 1$, we seek to report source IPs such that a source IP which contacts more than k unique destination IPs is reported with probability at least $1 - \delta$ while a source IP with less than k/b distinct destinations is (falsely) reported with probability at most δ. For example, when $k = 500$, $b = 2$ and $\delta = 0.05$, we want to report any source that contacts at least 500 distinct destinations and report no source that contacts less than 250 distinct destinations with probability 0.95.

We envision our algorithms to be useful in applications where it is acceptable to report sources whose distinct destination count is within a factor of 2 (or a factor of 5, 10, etc.) of a superspreader. For example, if we wish to identify sources involved in fast worm propagation and choose $k = 500$, it suffices to set $b = 2$, as we do not expect to find many sources (in normal traffic) that contact over 250 destinations within a short period. When a much finer distinction needs to be made (when b approaches 1), we will require a very high sampling rate, and there will not be a substantial reduction in memory usage or computational time.

Also, note that by our problem statement, a source will be identified as a superspreader with high probability when it has contacted between $\frac{k}{b}$ and k destinations. Thus, we will expect to report the (potential) k-superspreader *before* it has contacted k destinations, and so our approach will not delay the identification of the superspreader.

Table 1 summarizes the notation used in this paper.

2.2 Related Work and Previous Approaches

There has been a volume of work done in the area of streaming algorithms (see the surveys in [2, 28]). However, none of this work addresses the problem of identifying superspreaders efficiently. Perhaps most closely related is the problem of counting the number of distinct values in a stream. It has been studied by a number of papers (*e.g.*, [1, 3, 4, 10, 12, 18, 20, 21, 17]). The seminal algorithm by Flajolet and Martin [18] and its variant

due to Alon, Matias and Szegedy [1] estimate the number of distinct values in a stream up to a relative error of $\epsilon > 1$. Cohen [9], Gibbons and Tirthapura [20], and Bar-Yossef et al. [4] give distinct counting algorithms that work for arbitrary relative error. More recently, Bar-Yossef *et al.* [3] improve the space complexity of distinct values counting on a single stream, and Cormode *et al.* [10] 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 [21] give an (ϵ, δ)-approximation scheme for distinct values counting over a *sliding window* of the last N items, using $B = O(\frac{1}{\epsilon^2} \log(1/\delta) \log N \log R)$ memory bits. The algorithm extends to handle distributed streams, where B bits are used for each stream.

Previous Approaches: We now discuss existing approaches that may be applied to find superspreaders and their deficiencies.

- *Approach 1:* As a first approach, Snort [30] 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 keep an approximate count of distinct destinations a source contacts for each source [17]. Along these lines, Estan et al. [17] 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 $n \cdot S$ space where n is the total number of distinct sources (which can be $\Omega(N)$) and S is the amount of space required for the distinct counting algorithm to estimate its count. These approaches are particularly inefficient when the number of k-superspreaders is small and many sources contact far fewer than k destinations.
- *Approach 3:* The recent work by Weaver et al. [31] proposes an interesting data structure for finding scanning worms using Threshold Random Walk [23]. This data structure may be adapted to find superspreaders by tracking the number of distinct destinations contacted in the address cache. [1] However, it may not scale well to high-speed links, as it needs to keep some state for every flow for a period of time (and thus, the memory usage could be $\Omega(N)$). We present a concrete example for the parameters in [31]. The 1 MB connection cache keeps per-flow details, and after seeing 1 million flows (in one direction), fewer than 37% of new flows (in the same direction) are expected to map to an unused entry in the cache.[2] When new flows map into an existing entry in the connection cache, the counter for the source does not get updated. Thus, roughly 63% of superspreaders that appear after these million flows will not be identified (in expectation). With a time-out of 10 minutes, a rate of 1700 flows a second will saturate the 1 MB connection cache to this point. If we assume that these million flows come from distinct sources, and we need to find 1000-superspreaders with $b = 2$, and error probability $\delta = 0.05$, our two-level filtering algorithm needs only an expected 24KB of space. Thus, in this scenario, our algorithm is more accurate and requires much less space. However, their data structure is designed to find small scans quickly, and in this case performance of our algorithms will degrade.
- *Approach 4:* 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 D.

Other Related Work: A number of papers have proposed algorithms for related problems in network traffic analysis. Estan and Varghese [16] propose two algorithms to identify the large flows in network traffic, and give an accurate estimate of their sizes. Estan et al. [15] present an offline algorithm that computes the multidimensional traffic clusters reflecting network usage patterns. Duffield et al. [14] show that the num-

[1] We refer the reader to [31] for an understanding of the data structure. Here, we just describe how to use it for detecting superspreaders and what the issues with doing so are.

[2] Theorems on occupancy problems give these numbers. For details, see [27].

ber and average length of flows may be inferred even when some flows are not sampled, and compute the distribution of flow lengths. Golab et al. [22] present a deterministic single-pass algorithm to identify frequent items over sliding windows. Cormode and Muthukrishnan [11] present sketch-based algorithms to identify large changes in network traffic.

3 Algorithms for Finding Superspreaders

We now propose two efficient algorithms to find superspreaders. We first propose a one-level filtering algorithm, based on sampling from the set of *distinct* source-destination pairs. We then present a more complex algorithm based on a novel two-level filtering scheme, which will be more space-efficient than the one-level filtering algorithm for the distributions that (we expect) will be more common.

3.1 One-level Filtering Algorithm

The intuition for our one-level filtering algorithm for identifying k-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 m distinct destinations is expected to occur pm times in the sample. If p were $\frac{1}{k}$, then any k-superspreader (with its $m \geq k$ distinct destinations) would be expected to occur at least once in the sample, whereas sources that are not k-superspreaders would be expected *not* to occur in the sample. In this way, we may hope to use the sample to identify k-superspreaders[3].

There are several difficulties with this approach. First, the resulting sample would be a mixture of k-superspreaders and other sources that got "lucky" to be included in the sample. If there are no k-superspreaders, for example, the sample will consist only of lucky sources. To overcome this, we set p to be a constant factor c_1 larger than $\frac{1}{k}$. Then, any k-superspreader is expected to occur at least c_1 times in the sample, whereas lucky sources may occur a few times in the sample but nowhere near c_1 times. To minimize the space used by the algorithm, we seek to make c_1 as small as possible while being sufficiently large to distinguish k-superspreaders from lucky sources. A second, related difficulty is that there may be "unlucky" k-superspreaders that fail to appear in the sample as many times as expected. To overcome this, we have a second parameter $r < c_1$ and report a source as a k-superspreader as long as it occurs at least r times in the sample. A careful choice of c_1 and r is required.

$$c_1 = \begin{cases} \ln(1/\delta) \cdot \left(\frac{3b+2b\sqrt{6b}+2b^2}{(b-1)^2}\right) \\ \qquad \text{if } b \leq 3 \\ \ln(1/\delta)\max(b, 2/(1-\frac{e}{b})^2) \\ \qquad \text{if } 3 < b < 2e^2 \\ 8\ln(1/\delta) \text{ if } b \geq 2e^2 \end{cases} \quad (1)$$

$$r = \begin{cases} \frac{c_1}{b} + \sqrt{\frac{3c_1}{b}\ln(1/\delta)} & \text{if } b \leq 3 \\ \frac{ec_1}{b} & \text{if } 3 < b < 2e^2 \\ \frac{c_1}{2} & \text{if } b \geq 2e^2 \end{cases} \quad (2)$$

Figure 2. The parameters c_1 and r for the one-level filtering scheme.

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.

Algorithm Description: Let *srcIP* and *dstIP* be the source and destination IP addresses, respectively, in a packet. Let h_1 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 $h_1(\text{srcIP, dstIP}) < \frac{c_1}{k}$, where c_1 is given in Figure 2. This is determined by the analysis in Appendix B.
- Report all srcIPs with more than r retained, where r is given by the equations in Figure 2(b).

We can implement the algorithm above using two hash-tables (with $\frac{c_1 N}{k}$ buckets each): the first one to detect and discard duplicate pairs from the sample, and the second one to count the number of distinct destinations for each source in the sample.

In more detail, the above steps can be implemented as follows. Our implementation has the desirable property

[3]Note that we are sampling from the set of distinct source-destination pairs, not the set of packets we see; we perform a computation on every element in the stream – the "sampling" is at a conceptual level. The lower bounds of sampling approaches on counting distinct values [7] thus do not apply to our approach.

that each k-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_2 that maps srcIPs to $[0, 1)$.

- *Initially:* Let T_1 be a hash table with $c_1 N/k$ entries, where each entry contains an initially empty linked list of (srcIP, dstIP) pairs. Let T_2 be a hash table with $c_1 N/k$ 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 $h_1(s, d) \geq c_1/k$ then ignore the packet. Otherwise:

  1. Check entry $\frac{c_1 N}{k} \cdot h_1(s, d)$ 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 $\frac{c_1 N}{k} \cdot h_2(s)$ 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, m) with $(s, m + 1)$. If the new count equals $r + 1$, we report s. In this way, each declared k-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 k/c_1).

Analysis: The algorithm presented above will report any k-superspreader, and will not report a source that sends to less than $\frac{k}{b}$ distinct destinations with probability at least $1 - \delta$. The total space required in expectation is $O(c_1 N/k)$ words, while the per-packet processing time is constant with the hash-table implementation sketched above. Thus, for the typical case where δ is a constant and $b \geq 2$, the algorithm requires space for only $O(N/k)$ memory words. We give the precise theorem statement and the overhead analysis for this algorithm in Appendix B. We now give some examples to illustrate the one-level filtering algorithm.

Example: In this example, we set $k = 1000$ and $b = 2$, which means we are interested in reporting all sources that contact 1000 or more destinations within a given time period, without reporting any source that contacts less than 500 destinations within that time. Let N be the total number of packets seen in this time period.[4] For this, we find numerically that $c_1/k = 0.052$, and $r = 39$ suffice, when $\delta = 0.05$. Note that this sampling rate implies that in expectation, 94.8% of the packets will simply require one computation (hashing to see if the source-destination pair falls below c_1/k), and 5.2% of the packets will be selected for more processing. To store the source-destination pairs with a hash-table of $0.052N$, each of these selected packets will require (in expectation) no more than a read and a write of two IP addresses, which is a small computational overhead. To count the number of distinct destinations for any source in the first hash-table, we could use another hash-table and have an additional overhead of (at most) 2 reads and 2 writes (an IP address and a counter) per stored packet.

Note that these quantities do *not* depend on the distribution of the number of distinct destinations by source. That is, even if nearly every source sent to exactly one destination, the basic algorithm would have us store 5.2% of these sources, where the number 0.052 depends on k, b, and N. We would like to reduce the memory used in storing these non-superspreader sources. Further, we expect that most traces will have a very large number of sources that contact only a few distinct destinations, and *very* few superspreaders. Can we track the superspreaders accurately without tracking so many non-superspreaders?

The difficulty here is that the one-level filtering algorithm needs a certain minimum sampling rate in order to distinguish between sources that send to k destinations and $\frac{k}{b}$ destinations. But sources that contact only a few destinations also get sampled *at this rate*. In the next section, we will effectively reduce the sampling rate of these non-superspreader sources without compromising on the accuracy of the algorithm for detecting superspreaders.

3.2 Two-Level Filtering Algorithm

We now present another algorithm that uses *two* levels of filters and is more memory-efficient than one-level filtering in most cases. At a high level, the algorithm uses two levels of filtering in the following manner: the first-level filter effectively decides whether we should keep more information about a particular source, while the second-level filter 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 only 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

[4] In a real setting, N could be determined historically.

```
function Two-Level Filtering(s, d)
    Level2(s, d); Level1(s, d);

function Level1(s, d)
    if(h1(s,d) < r1) insert s into T1

function Level2(s, d)
    if (h2(s, d) > r2) return;
    if (s ∉ T1) return;
    else compute p = h2(s,d)/r2 · γ and insert s
      into T2,p.

function Output
    output all sources that appear in more
      than ω of the hash-tables T2,i.
```

Figure 3. Two-level filtering pseudocode, where (s, d) represents a source-destination pair.

use it for sources that pass the first filter.

Intuitively, the reason why the two-level filtering algorithm is more space-efficient than the one-level filtering algorithm is because the sampling rate for the *first level* of two-level filtering algorithm is lower than the sampling rate of the one-level filtering algorithm. (To compensate, the sampling rate for the *second level* will need to be a bit higher.) If a source contacts sufficiently many destinations, it will be sampled (and thus, stored) in both the one-level filtering algorithm and the two-level filtering algorithm. But if a source contacts only a few destinations, the probability that it is sampled (and tracked) in the two-level filtering algorithm is much lower than the probability that it is sampled (and tracked) in the one-level filtering algorithm. Thus, the two-level filtering algorithm will store fewer sources that contact very few distinct destinations. 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.

Algorithm Description: The algorithm takes r_1, r_2, γ, ω as parameters, where r_1 and r_2 represent the sampling rate in the first and second level respectively, and ω is a threshold. Given the required values for k and b, the values of r_1, r_2, γ, ω may be determined as in the analysis of Theorem C.1.

We keep one hash-table T_1 at the first level, and γ hash-tables denoted $T_{2,i}$ at the second level. Let h_1 and h_2 be uniform random hash functions that take a source-destination pair and return a value in $[0, 1)$ as described in the previous section.

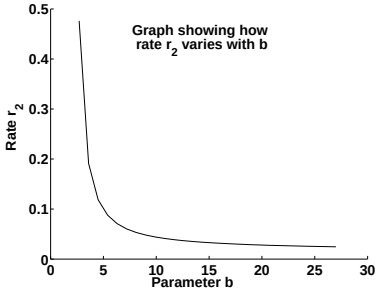


Figure 4. The rate r_2 required for $k = 1000$, $\delta = 0.01$, with varying b.

For each packet (s, d), the network monitor performs the following operations as given in pseudocode in Figure 3:

- Step 1: First, we compute $h_2(s, d)$. If $h_2(s, d)$ is greater than rate r_2, we skip to step 2. Otherwise, we check to see if the source s is present in the hash-table T_1. If s is not present in T_1, then again, we skip to step 2. Otherwise, we insert s into level 2 as follows: If $h_2(s, d) < r_2$ and s is present in T_1, we insert s into the level-2 hash-table $T_{2,p}$, where $p = \frac{h_2(s,d)}{r_2} \cdot \gamma$. Thus, we insert s into level 2 with at most probability r_2, and every source appearing in level 2 appears in level 1.
- Step 2: If $h_1(s, d)$ is less than rate r_1, we insert s into T_1.

Finally, we output all sources that appear in more than ω of the tables $T_{2,i}$.

Optimizations: Note that in the above description, we use hash-tables 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 hash-tables to store the sampled elements. A discussion of Bloom filters may be found in [5, 6]. In addition, in the above description, we chose the probability of inserting a sampled packet into any level-2 hash-table $T_{2,i}$ to be equal to $1/\gamma$, for simpler description and analysis. We can easily generalize this to alter the probability of inserting a sampled packet into any level-2 hash-table to be non-uniform, e.g., an exponential distribution.

Analysis: We give a summary of the analysis results here, and defer the theorem statement and details to Appendix C. When the parameters of the two-level filtering algorithm are chosen appropriately (as shown in the analysis of Theorem C.1), it reports srcIPs such that any k-superspreader is reported with probability at least

$1-\delta$, while a srcIP with less than k/b distinct destinations is (falsely) reported with probability less than δ. Figure 4 shows how the required rate r_2 varies with b. (The figure is not meant to illustrate exact values of r_2 that would be used in experiments, but just to give an intuition of how r_2 varies with b for a fixed k and δ.) The threshold ω and the number of hash-tables γ vary similarly. The expected space required by is $O(r_1 N + r_2 N)$. Note that, for a fixed $b \geq 2$, both r_1 and r_2 are $O(\frac{1}{k} \ln \frac{1}{\delta})$, and thus the space required is $O(\frac{N}{k} \ln \frac{1}{\delta})$.[5]

We may make a similar statement when we use Bloom filters rather than hash-tables 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 accuracy 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. Also, in practice, we optimize our choice of the parameters numerically for both algorithms, since the bounds given by the theorems may have larger constant factors than are strictly necessary.

Example: In this example, we set $k = 1000$ and $b = 2$, which means we are interested in reporting all sources that contact 1000 or more destinations, without reporting any source that contacts less than 500 destinations. For this, we find numerically that $r_1 = 0.006$, $r_2 = 0.15$ and $\gamma = 100$ suffice, when $\delta = 0.05$. Note that this sampling rate implies that 85% of the flows will need only to be hashed once and incur no memory accesses, and 15% of the flows will have to be additionally processed. The amount of computational overhead that these selected flows incur will depend on the number of distinct destinations that their respective source contacts, so we will examine two specific cases:

Case 1: For the sources that contact exactly one distinct destination each, in expectation, 0.6% of the packets will be entered into the first level, and require 1 read and 1 write (of one IP address), and 15% of the packets will require exactly 1 read.

Case 2: For any particular superspreader, at most 15% of the distinct flows (corresponding to that superspreader) will require 2 distinct memory locations (1 in level-1, 1 in level-2, of 1 IP address each, with 2 reads and 1 write).

Note that if the trace contains only sources that contact one destination each (the first case), the two-level filtering algorithm has much less overhead than the one-level filtering algorithm, and if the trace contains only superspreaders (the second case), two-level filtering algorithm has about three times as much overhead as one-level filtering algorithm. This gives an idea of the trade-off between the algorithms; we expect that most sources only contact a few distinct destinations, and thus the traces will resemble the first case far more than the second case.

Using a Bloom filter in both levels will reduce the memory storage required, but increase the number of accesses that need to be made. If we use a Bloom filter with 8 independent hash functions at each of the two levels, our memory storage will drop by a constant factor of at least 2.5 (estimating conservatively to account for additional false positives), and our computational overhead will increase by a factor of 8 – since we will need to make 8 memory accesses for every memory access of the hash-table implementation.

Note that there could exist as many as $\frac{N}{k}$ superspreaders; thus, for constant b, all our bounds are within a $\log \frac{1}{\delta}$ factor of the asymptotically optimal values.

4 Extensions

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

4.1 Superspreaders with Deletion

We can 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 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 the one-level filtering algorithm (Section 3.1), a deletion of (s, d) is handled by first checking to see if

[5] For the two-level filtering algorithm as stated here, the sampling rates r_1 and r_2 may not always exist for a given k, b and δ. (The possible values of k and b are a function of the confidence level δ; as δ is decreased, sufficiently small k and k/b may not be distinguishable in this algorithm.) To address this issue, one could modify the two-level filtering algorithm by running it c times in parallel. However, this issue does not come up for reasonable values of k, b and δ (and this is also seen in our experiments), and so we omit further discussion of this issue.

(s, d) is in the hash-table. 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 (s, d) from the hash-table. The precision, space, and time bounds are the same as in the case without deletions. Similarly, we can extend the hash-table implementation of the two-level filtering algorithm to handle deletions as well.[6]

We can also use this approach to find those sources which have more than k failures and fewer than k/b successes. We could find these sources by computing separately the sources that have at least k successes and those that have at least k failures, and return the appropriate difference.

Note that the definition of a k-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 k-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, the one-level filtering algorithm can report a source as a *tentative* k-superspreader when its count in T_2 reaches $r + 1$, and then report at the end of the interval which sources (still) have counts greater than r.

4.2 Superspreaders over Sliding Windows

In this section, we show how to extend our algorithms to handle sliding windows. Our goal is to identify k-superspreaders with respect to the most recent W packets, i.e., hosts which contact more than k 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.

Figure 5 gives an example of a stream subject to a sliding window of size $W = 4$. The top row shows the packets in the sliding window after the arrival of $(s1, d3)$. The middle row shows that on the arrival of $(s2, d3)$, the window includes this pair but drops $(s1, d1)$. The bottom row shows that on the arrival of $(s2, d1)$, the window adds this pair but drops $(s1, d2)$.

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 4.1 where we are given at the time of deletion the source-destination pair to delete.

[6]Technically, we need a slight modification of the algorithm described earlier; we need to store the destinations as well at each level-2 hash-table; this may increase the memory required by at most a factor of 2.

$(s1, d1)$, $(s1, d2)$, $(s2, d2)$, $(s1, d3)$
$(s1, d2)$, $(s2, d2)$, $(s1, d3)$, $(s2, d3)$
$(s2, d2)$, $(s1, d3)$, $(s2, d3)$, $(s2, d1)$

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 $(s1, d3)$. The middle row shows that on the arrival of $(s2, d3)$, the window includes this pair but drops $(s1, d1)$. The bottom row shows that on the arrival of $(s2, d1)$, the window adds this pair but drops $(s1, d2)$.

In the sliding window setting, a source may transition between being a k-superspreader and not, as the window slides. In Figure 5, for example, suppose that the threshold for being a k-superspreader is having at least 3 distinct destinations (e.g., $k = 3$). Then source $s1$ is a superspreader in the first window, but not the second or third windows.

We show how to adapt one-level filtering algorithm to handle sliding windows. the approach for two-level filtering algorithm is similar. 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 $x-W+1, \ldots, x$. At a high level, the algorithm works by (1) maintaining the pairs in our sample sorted by sequence number, in order to find in $O(1)$ 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 $O(1)$ 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_1 and T_2 be as in the original one-level filtering algorithm, except that T_1 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 $\frac{c_1 N}{k} \cdot h_1(s, d)$ of T_1. If the triple exists, then because T_1 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_1.

(b) Use $\frac{c_1 N}{k} \cdot h_2(s)$ to look-up s in T_2, and decrement the count of this entry in T_2, i.e., replace the pair (s, m) with $(s, m-1)$.

(c) If the new count equals 0, we know that the source no longer appears in the sample and we remove the pair from T_2.

On the other hand, if the triple does not exist, then there is some other triple (s, d, n') 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 $h_1(s, d) \geq c_1/k$ ignore the packet. Else:

(a) Check entry $\frac{c_1 N}{k} \cdot h_1(s, d)$ of T_1 for a triple with s and d. If such an entry exists, replace it with (s, d, x), 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 s, i.e., this is the first time (s, d) has appeared in the window. We use $\frac{c_1 N}{k} \cdot h_2(s)$ 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, m) with $(s, m+1)$. If the new count equals $r+1$, we report s.

The precision, time and space bounds are the same as in one-level filtering algorithm 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.

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 6. Example streams at 3 monitoring points

4.3 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 6 and $k = 3$. 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 k-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 6 is an example of this. A key challenge is to enable this distributed identification while having only limited communication between the monitoring points.

We describe how to modify our one-level filtering algorithm to work in a distributed setting. First, each network monitor runs the algorithm as described in Section 3.1 (all using the same hash function, and with appropriately sized hash-tables, say $c_1 N/kj$ if each of the j monitors expects to see N/j packets). The monitor reports any locally detected superspreader. Next, at the end of the stream, each monitor sends its hash-table of (s, d) pairs to the central monitor. Finally, the central monitor treats these hash-tables as a stream of (s, d) pairs, and using the same hash function, runs the algorithm on this stream, and reports any superspreader found.

The overall space and time overhead of first step above summed over all the monitors is the same as if one monitor monitors the union of the streams. The second step requires a total amount of communication equal to the sum of the space for the hash-tables, i.e., an expected $O(c_1 N/k)$ memory words. Accounting for the last step 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 N packets in all, the algorithm achieves the precision bounds given in Theorem B.1. Thus our distributed one-level filtering algorithm uses little memory and little communication. On the other hand, a similar extension to the two-level filtering algorithm results in more communication in the distributed setting – specifically, at each step, the monitors would need to have access to all the (individual) first-level hash-tables, which results in significant increase in communication between

monitors.

5 Experimental Results

We implemented our algorithms for finding superspreaders, and we evaluated them on network traces taken from the NLANR archive [29], 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 *SHA1* 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. Both algorithms are implemented so that the superspreaders get output at the end of the run, once all the packets have been processed. We ran our experiments on several traces and obtained 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 2.2.

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 Figure 8), we inserted various attack packets where some sources contacted a high number of distinct destinations. That is, for given parameters k and b, we added 100 sources that send to k destinations each, and 100 sources that send to just under 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 $\delta = 0.05$. In Figure 7, 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 of the number of distinct destinations each source contacts. We optimize our choice of the other parameters numerically for both algorithms (in a manner suggested by the analysis of the theorems), since the bounds given by the theorems may have larger constant factors than are strictly necessary.

We observe that the accuracy of both algorithms is comparable and bounded by δ, which confirms our theoretical results. Note that using a smaller value of δ would produce a smaller false positive rate and false negative rate. 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 δ. When a source sends to a very small number of destinations (much smaller than k/b), the probability that it becomes a false positive is significantly lower than δ. Likewise, when a source sends to a very large number of destinations ($\gg k$), the probability that it becomes a false negative is much less than δ. 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 δ. Further, *all of the false positives in our experiments come from the sources at the threshold that we added*, not the original trace itself. The false positive rate is typically of much more importance than the false negative rate, since there are usually many more sources that could be false positives than sources that could be false negatives. Thus, it is very useful to verify that the false positive rate is much lower than the stated δ in real traces, and that the false positives observed do come only from the inserted traffic at the threshold.

k	b	False Positives		False Negatives	
		1-Level	2-Level	1-Level	2-Level
500	2	8.1e-5	6.3e-5	0	0
500	5	1.13e-4	1.13e-4	0	0
500	10	1.35e-4	8.1e-5	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	1.13e-4	9.45e-5	0	0.03
5000	2	8.1e-5	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

Figure 7. Evaluation of the precision of one-level filtering and two-level filtering algorithms over various settings for k and b, with $\delta = 0.05$.

5.2 Memory usage on long traces

We now examine memory used on very long traces by one-level filtering algorithm (Section 3.1) and the hash-table and Bloom-filter implementations of the two-level filtering algorithm (Section 3.2). To distinguish the two implementations of the two-level filtering algorithm, we will refer to the hash-table implementation as 2LF-T, and the Bloom-filter implementation as 2LF-B. We will

	Length (in sec)	No. distinct sources	No. distinct src-dst pairs	N (no. of packets)
1	65	59,862	194,060	$2.88e6$
2	154	282,484	416,730	$3.09e6$
3	207	$1.21e6$	$1.35e6$	$4.02e6$
4	269	$2.12e6$	$2.29e6$	$4.49e6$

Figure 8. Base traces used for experiments

use 1LF to refer to the one-level filtering algorithm. We examine the memory used as the parameters k, b and N are allowed to vary. The memory usage reported is the number of elements actually stored, which is always very close to the size of the hash-tables. (The size of each hash-table set to be the expected number of elements that will be inserted, based on the sampling rates and N.) For the bloom filter implementation, we use 8 independent hash functions.

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 Figure 8. 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 9 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 9(a), 9(b), and 9(c) contain the same number of distinct sources, even though the value of b differs.

We observe that the memory used by the two algorithms is strongly correlated with b, 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 2LF) from Figure 4, in section 3.2.

Another observation is that, as expected, the memory used by 1LF eventually exceeds the memory used by 2LF-T & 2LF-B, for every value of b. The number of sources at which the memory used by 1LF exceeds the memory used by 2LF-T & 2LF-B also depends on b. We also note that, as expected, the memory used by 2LF-B is much less than the memory used by 2LF-T and 1LF .

We next examine the memory usage as k changes, which is shown in Figures 10 and 11. We observe that the total memory used drops sharply as k increases, as expected: in 10(a), at $k = 500$, the memory used ranges from $20,000$ to $200,000$ IP addresses; in 10(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 k, as the number of packets N increases, the memory used by 1LF eventually exceeds the memory used by 2LF-T & 2LF-B. This is because of the two-level sampling scheme. Since the first sampling rate r_1 is much smaller than c_1/k in 1LF , the number of non-superspreader sources stored in 2LF-T & 2LF-B ($r_1 N$ in expectation) is much less than in 1LF . The actual number of sources at which this occurs depends on k. As k increases, the number of sources at which the memory used by 1LF exceeds the memory used by 2LF-T (and 2LF-B) also increases, since the sampling rates for both algorithms decrease in the same way. We also observe that, once again, the memory used by 2LF-B is significantly lower than 1LF and 2LF-T.

The graphs in Figure 11 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.

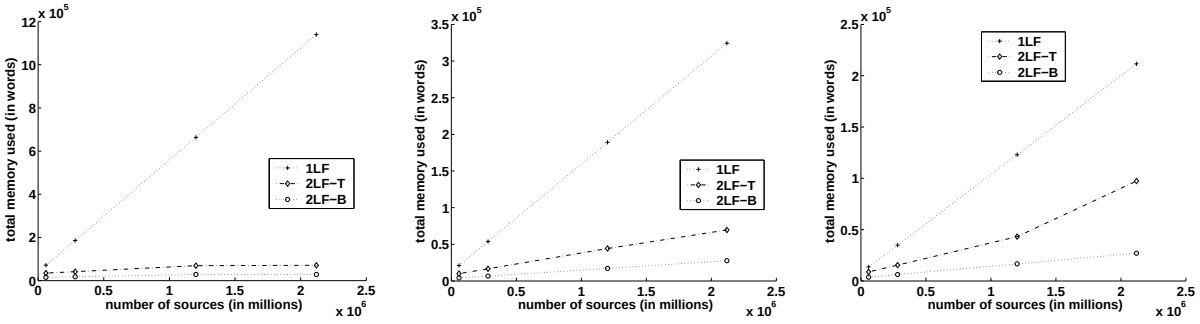
Lastly, we tested both 1LF and 2LF-T on a trace with 10 million sources that contacted a few destinations each. At $k = 1000$ and $b = 2$, the memory usage of 1LF and 2LF-T were about 1.04 million and 60,100 IP addresses respectively. Thus, we see that our algorithms do indeed scale well as the number of flows increases.

5.3 Comparison with an Alternate Approach

We now show results comparing our approach to the Approach 4 described in Section 2.2: 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 D. We do not show experimental comparisons with the other approaches as they all keep per-flow state, and so, it is clear that they need far more space than our algorithms.

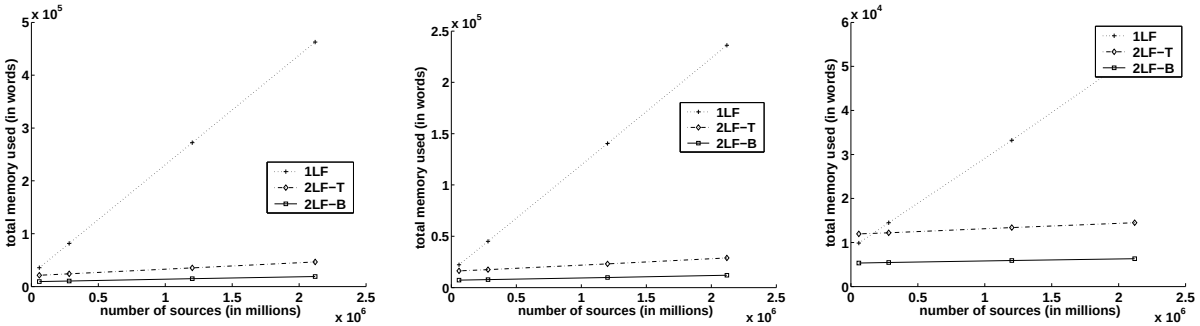
We chose the parameters for LossyCounting and the distinct counting algorithm so that (a) the memory usage was minimized for each b and (b) they produced the false positive rates similar to our algorithms over 10 iterations. We show experimental results with two variants of the approach: (1) use one 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.

Figure 12 shows the result of the comparison of memory usage at $k = 1000$, for $b = 2, 5$ and 10, on Trace



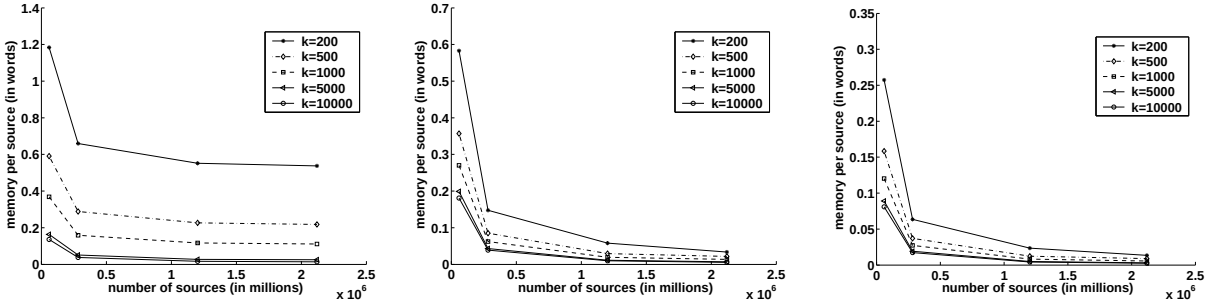
(a) $b = 2$ (b) $b = 5$ (c) $b = 10$

Figure 9. 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 10. Total memory used by the algorithms in words (i.e., IP addresses) vs number of distinct sources, for $k = 500, 1000$ and 5000, at $b = 2$.



(a) One-level filtering (b) Two-level filtering (hashtable) (c) Two-level filtering (Bloom filters)

Figure 11. Memory used per source vs number of distinct sources, for all k, by 1LF , 2LF-T, & 2LF-B at $b = 2$.

b	1LF	2LF-T	2LF-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

Figure 12. Comparisons of total memory used with traces 1 & 2 for $k = 1000$ and varying b.

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 1LF when $b = 2$. We explain why Alt I is better than Alt II in Appendix D.

6 Conclusion

In this paper, we have described new streaming algorithms for identifying superspreaders on high-speed networks. Our algorithms give proven guarantees on the accuracy and the memory requirements. Compared to previous approaches, our algorithms are substantially more efficient, both theoretically and experimentally. We also provide several extensions to our algorithms – we can identify superspreaders in a distributed setting, over the sliding windows, and when deletions are allowed in the stream (which lets us identify sources that make a large number of failed connections to distinct destinations). 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 network monitoring.

7 Acknowledgements

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. We also gratefully acknowledge support from the National Science Foundation grant number CNS-0433540 entitled "CyberTrust Center: Security Through Interaction Modeling (STIM)". 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.

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, Sept. 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)*, Jan. 2002.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

[7] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. 19th ACM Symp. on Principles of Database Systems*, pages 268–279, May 2000.

[8] 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.

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

[10] 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, Aug. 2002.

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

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

[13] 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.

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

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

[16] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of SIGCOMM'02*, 2002.

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

[18] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Computer and System Sciences*, 31:182–209, 1985.
[19] 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.
[20] P. B. Gibbons and S. 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.
[21] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, Aug. 2002.
[22] 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.
[23] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.
[24] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
[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] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.
[28] S. Muthukrishnan. Data streams: Algorithms and applications. Technical report, Rutgers University, Piscataway, NJ, 2003.
[29] NLANR. National laboratory for applied network research. http://pma.nlanr.net/Traces/, 2003.
[30] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.
[31] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th Usenix Security Conference*. USENIX, August 2004.

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 B.1

Accuracy Analysis: Our analysis yields the following theorem for precision:

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

A proof follows the discussion of the overhead analysis.

Overhead Analysis: The total space is an expected $O(c_1 N/k)$ memory words. The choice of c_1 depends on b. By equation 2, 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 $O\left(\frac{N}{k}\ln\frac{1}{\delta}(1+\frac{1}{(b-1)^2})\right)$ bits. For the typical case where δ is a constant and $b \geq 2$, the algorithm requires space for only $O(N/k)$ memory words.

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

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 less than r successes is less than δ, and

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

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

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 $\sqrt{b} < \sqrt{6}$, and hence $c_1 > 3b\ln(1/\delta)$. It follows that for the c_1 in equation 6, P1 and P2 hold when $r = (1+\epsilon_2)(c_1/b) = \frac{c_1}{b} + \sqrt{\frac{3c_1}{b}\ln(1/\delta)}$.

Next consider the case when $3 < b < 2e^2$. As argued above, P1 holds with $\epsilon_1 = \sqrt{\frac{2}{c_1}\ln(1/\delta)}$, as long as $c_1 > 2\ln(1/\delta)$. (This analysis did not depend on b.) 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 $b = 4$, the c_1 in equation 6 is less than $3b\ln(1/\delta)$. Instead, we apply equation 3 with $\beta = e$: $Pr[X_2 \geq ec_1/b] \leq e^{(1-1/e-1)ec_1/b} = e^{-c_1/b}$. We require that $r = ec_1/b = (1-\epsilon_1)c_1$, i.e., $\epsilon_1 = 1 - e/b = \sqrt{\frac{2}{c_1}\ln(1/\delta)}$. Solving for c_1, we get $c_1 = 2\ln(1/\delta)/(1-\frac{e}{b})^2$. Moreover, P1 holds for all $c_1 \geq 2\ln(1/\delta)/(1-\frac{e}{b})^2$.

Similarly, setting $e^{-c_1/b} = \delta$ and solving for c_1, we get that $c_1 = b\ln(1/\delta)$ and moreover, P2 holds for all $c_1 \geq b\ln(1/\delta)$. Thus selecting c_1 such that

$$c_1 = \ln(1/\delta)\cdot\max(b, 2/(1-e/b)^2) \quad \text{when } 3 < b < 2e^2 \quad (7)$$

implies both P1 and P2 hold when $r = ec_1/b$. (Note that $0 < (1-e/b)^2 < 1$ and hence $c_1 > 2\ln(1/\delta)$, as required for P1.)

Finally, consider the case when $b \geq 2e^2$. Note that the c_1 from equation 7 grows linearly in b. Thus for large b, we seek a modified analysis in which c_1 does not grow asymptotically with b. First we apply equation 3 with $\beta = b/2$: $Pr[X_2 \geq (b/2)(c_1/b)] \leq e^{(1-2/b-\ln(b/2))c_1/2} < e^{-c_1/2}$ because $b \geq 2e^2$ implies $\ln(b/2) \geq 2$ implies $1 - 2/b - \ln(b/2) < -1$. Thus $r = c_1/2 = (1-\epsilon_1)c_1$, i.e. $\epsilon_1 = 1/2$. By equation 5, we have $Pr[X_1 \leq r] \leq e^{-(1/2)^2 c_1/2} = e^{-c_1/8}$. It follows that selecting $c_1 = 8\ln(1/\delta)$ when $b \geq 2e^2$ implies that both P1 and P2 hold when $r = c_1/2$. □

C Two-level Filtering Algorithm: Analysis

For ease of explanation, we present an analysis of a simpler version of the two-level filtering algorithm in Section 3.2. In this version, every time a source must be inserted into level-2, we let each hash-table decides independently whether the source gets inserted in that table. Using the notation of Section 3.2, here is step 1 the algorithm: each level-2 hash-table $T_{2,i}$ has a hash function $h_{3,i}$ associated with it. if $h_2(s,d) < r_2$ and s is present in T_1, we consider inserting s into each level 2 hash-tables $T_{2,i}$ separately. For each $T_{2,i}$, we check if $h_{3,i}(s,d)$ is less than $1/\gamma$. If $h_{3,i}(s,d) < 1/\gamma$, we insert s into $T_{2,i}$. Note that, thus, we may insert s into multiple level-2 hash-tables, but we expect to insert into exactly one level-2 hash-table.

The following proof for the algorithm above can be adapted to the two-level filtering algorithm in Section 3.2 by using martingale tail bounds on occupancy problems [27] instead of Chernoff bounds.

Theorem C.1. *Given k, $N, b > 1$ such that $k/b > 1$ and $0 < \delta < 1$. Let $z = \max(\frac{2b}{b-1}, 5)$. Let $r_1 = \frac{z}{k}\log\frac{2}{\delta}$, and r_2 be minimal value that satisfies the following constraints:*

$$r_2 \geq \frac{2\ln 2/\delta}{k(1-e^{-(z-1)/z})\epsilon_1^2},$$

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

where $\epsilon_1 = 1 - \frac{1-e^{-2/b}}{1-1/e^{-(z-1)/z}}(1+\epsilon_2)$,

$\epsilon_2 > 0$, *and* $0 < \epsilon_1 < 1$.

Also, $r_2 \leq 1$.

Let ϵ_1' be the value of ϵ_1 when r_2 is minimized in the above constraints. Let $\gamma = r_2 k$, and $\omega = (1-\epsilon_1')\gamma(1-\frac{1}{e})$.

Thus, for any given $b > 1$, positive $\delta < 1$, and k such that $k/b > 1$, if r_2 exists satisfying above constraints, Algorithm II reports srcIPs such that any k-superspreader is reported with probability at least $1-\delta$, while a srcIP with less than k/b distinct destinations is (falsely) reported with probability less than δ.

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.

All logarithms in the proof are base e.

False negatives. We analyze the false negative error in two parts. For a source i with $n_i > k$, we first show that the source is inserted into T_1, with probability $1 - \frac{\delta}{2}$, in the first $1/z$ fraction of its total pairs. Then, we show that it is inserted into at least ω of the tables $T_{2,i}$ over the rest of its pairs, with probability $1 - \frac{\delta}{2}$, conditioned on its presence in T_1. Together, these two parts ensure that, with probability at least $1 - \delta$, the source is present in ω of the tables $T_{2,i}$, after all the distinct destinations for that source are seen.

For the first part of the false negative analysis, we need to set the rate of sampling r_1 so that any source that appears more than k times in the sampled set will be sampled with a probability of $1-\frac{\delta}{2}$, within its first $\frac{1}{z}$ fraction of packets. This is equivalent to saying that any source s with $\frac{k}{z}$ packets is present in T_1 with probability at least $1 - \frac{\delta}{2}$. Equivalently, $1 - Pr[s \in T_1 | n_s \geq k/z] = (1 - r_1)^{\frac{k}{z}}$ which needs to be bounded by $\frac{\delta}{2}$. Therefore,

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

This gives us a lower bound on r_1. We want r_1 to be as small as possible to minimize the memory needed, so we set r_1 to its lower bound.

Now we analyze the second part of the false negative error. By setting r_1 as specified, we know that the source will be present in T_1 with probability $1 - \frac{\delta}{2}$ in the first $\frac{k}{z}$ packets. So, in the remaining $k(1 - \frac{1}{z})$ packets, we examine the probability that the source will fall into ω tables in the second step, conditioned on the event that the source is already present in T_1.

Let X_{ij} be the indicator variable for the event that source i is put into the jth table, $T_{2,j}$, when the source is already present in T_1. The expected number of tables $T_{2,j}$ that will contain source i is the sum $S_j = \sum_{j=1}^{\gamma} X_{ij}$.

Let $k' = k(1 - \frac{1}{z})$. For a source i with $n_i \geq k'$: $Pr[X_{ij} = 1] = 1 - Pr[X_{ij} = 0] \geq 1 - (1 - \frac{r_2}{\gamma})^{k'}$ which is at least $1 - e^{-r_2 k'/\gamma}$.

Therefore, for a source i with $n_i \geq k'$, we can write the expected value of tables set $E[S_i] = \sum_{j=1}^{\gamma} E[X_{ij}] \geq \gamma(1 - e^{-r_2 k'/\gamma})$. Let $\mu_1 = E[S_i]$. The random variable S_i is equivalently the result of sampling γ coins of bias $Pr[X_{ij} = 1]$, so we can use the Chernoff bounds to get a lower bound on S_i, with with probability at least $1 - \frac{\delta}{2}$.That is, for $\epsilon_1 \in (0, 1)$, by Chernoff bounds: $Pr[S_i \leq (1 - \epsilon_1)\mu_1] \leq e^{\frac{-\mu_1 \epsilon_1^2}{2}}$.

Let $\omega_1 = (1 - \epsilon_1)\mu_1$. We need to have $Pr[S_i \leq (1-\epsilon_1)\mu_1] \leq \frac{\delta}{2}$. We can get this by setting $e^{\frac{-\mu_1 \epsilon_1^2}{2}} \leq \frac{\delta}{2}$. Therefore, $\mu_1 \geq \frac{2}{\epsilon_1^2} \log \frac{2}{\delta}$. Since $\mu_1 \geq \gamma(1-e^{\frac{-r_2 k'}{\gamma}})$ we need to have $\gamma(1-e^{-r_2 k'/\gamma}) \geq \frac{2}{\epsilon_1^2} \log \frac{2}{\delta}$. Since $\gamma = r_2 k$, $r_2 k(1 - e^{-k'/k}) \geq \frac{2}{\epsilon_1^2} \log \frac{2}{\delta}$. Therefore,

$$r_2 \quad \geq \quad \frac{2}{k\epsilon_1^2(1 - e^{-k'/k})} \log \frac{2}{\delta}.$$

Thus, we get the first constraint in the optimization problem in Theorem C.1.

False positives. Now we analyze the false positive error. A source i is a false positive if $n_i < k/b$, 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 ω of the tables $T_{2,i}$ with sufficiently low probability. Therefore, we compute a constraint on the rate of sampling r_2, so that, with probability at least $1-\delta$, no more than ω tables will contain the source. We note that the probability of being a false positive is maximized when n_i is as large as possible, so we assume now that $n_i = k/b$.

As in our analysis for the false negatives, let X_{ij} be the indicator variable for the event that the source i is put into the jth table in T_2: X_{ij} is 1 if the source is put into the jth table, and 0 otherwise. Once again, we can compute the probability that source i gets put into the jth table $T_{2,j}$ as follows: $Pr[X_{ij} = 1] = 1 - Pr[X_{ij} = 0] \leq 1 - e^{-2r_2 k/b\gamma}$.[7]

Therefore, the expected number of tables containing source i with $n_i = \frac{k}{b}$ can be written as $E[S_i] = \sum_{j=1}^{\gamma} E[X_{ij}] \leq \gamma(1 - e^{-2r_2 k/b\gamma})$. Substituting for γ, we get $E[S_i] \leq r_2 k(1 - e^{-2/b})$.

Let $\mu_2 = E[S_i]$. Once again, with Chernoff bounds, we can get, for $\epsilon_2 > 0$, $Pr[S_i \geq (1 + \epsilon_2)\mu_2] \leq e^{-c\mu_2}$, where $c = -\log e^{\epsilon_2}(1 + \epsilon_2)^{-(1+\epsilon_2)}$

[7]The constant 2 can be changed depending on our assumption on the relationship between r_2 and γ. Here, for simplicity, we only assume $r_2 < \gamma$, which is true when $k > 1$.

Let $\omega_2 = (1+\epsilon_2)\mu_2$. Since we want to bound $Pr[S_i \geq (1+\epsilon_2)\mu_2]$ by δ, this becomes $\delta \geq e^{-\mu_2((1+\epsilon_2)\log(1+\epsilon_2)-\epsilon_2)}$. Substituting for μ_2, we get $\log\frac{1}{\delta} \leq r_2 k(1-e^{-2/b})((1+\epsilon_2)\log(1+\epsilon_2)-\epsilon_2)$. Thus,

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

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[8]:

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

Thus, we get the last relation in the problem.

We wish to minimize r_2, subject to these constraints, because the expected memory is $O(r_1 N + r_2 N)$. Therefore, the solution to the problem in Theorem C.1, gives us a sampling rate r_2 such that a source with k distinct destinations will be in at least ω tables, and a source sending to less than k distinct destinations will not be in ω tables, 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 $\log\frac{1}{\delta}$ distinct counters per source, and use their median for counting approximately the number of distinct elements (Alt II). The distinct counting algorithm requires $O(\log\frac{1}{\delta})$ 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 $\log\frac{1}{\delta}$ factor increase in the memory usage, even though the actual constant t decreases.

[8] Alternately, we could use a constraint $\omega_1 > \omega_2$, and set ω to be some value between ω_1 and ω_2.