Bitmap Algorithms for Counting Active Flows on High Speed Links

Cristian Estan, George Varghese, Mike Fisk
Computer Science and Engineering Department
University of California San Diego
cestan,varghese,mfisk@cs.ucsd.edu

ABSTRACT

This paper presents a family of bitmap algorithms that address the problem of counting the number of distinct header patterns (flows) seen on a high speed link. Such counting can be used to detect DoS attacks and port scans, and to solve measurement problems. Counting is especially hard when processing must be done within a packet arrival time (8 nsec at OC-768 speeds) and, hence, must require only a small number of accesses to limited, fast memory. A naive solution that maintains a hash table requires several Mbytes because the number of flows can be above a million. By contrast, our new probabilistic algorithms take very little memory and are fast. The reduction in memory is particularly important for applications that run multiple concurrent counting instances. For example, we replaced the port scan detection component of the popular intrusion detection system Snort with one of our new algorithms. This reduced memory usage on a ten minute trace from 50 Mbytes to 5.6 Mbytes while maintaining a 99.77% probability of alarming on a scan within 6 seconds of when the large-memory algorithm would. The best known prior algorithm (probabilistic counting) takes 4 times more memory on port scan detection and 8 times more on a measurement application. Fundamentally, this is because our algorithms can be customized to take advantage of special features of applications such as a large number of instances that have very small counts or prior knowledge of the likely range of the count.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*traffic measurement, counting active flows*

General Terms

Algorithms,Measurement

Keywords

Network traffic measurement, counting flows

1. INTRODUCTION

Internet links operate at high speeds, and past trends predict that these speeds will continue to increase rapidly. Routers and intrusion detection devices that operate at up to OC-768 speeds (40 Gigabits/second) are currently being developed. While the main bottlenecks (e.g., lookups, classification, quality of service) in a traditional router are well understood, what are the corresponding functions that should be hardwired in the brave new world of security and measurement? Ideally, we wish to abstract out functions that are common to several security and measurement applications. We also wish to study efficient algorithms for these functions, especially those with a compact hardware implementation.

Toward this goal, this paper isolates and provides solutions for an important problem that occurs in various networking applications: *counting the number of active flows among packets received on a link during a specified period of time.* A *flow* is defined by a set of header fields; two packets belong to distinct flows if they have different values for the specified header fields that define the flow. For example, if we define a flow by source and destination IP addresses, we can count the number of distinct source-destination IP address pairs seen on a link over a given time period. Our algorithms measure the number of active flows using a very small amount of memory that can easily be stored in on-chip SRAM or even processor registers. By contrast, naive algorithms described below would require massive amounts of memory necessitating the use of slow DRAM.

For example, a naive method to count source-destination pairs would be to keep a counter together with a hash table that stores all the distinct 64 bit source destination address pairs seen thus far. When a packet arrives with source and destination addresses say $< S, D >$, we search the hash table for $< S, D >$; if there is no match, the counter is incremented and $< S, D >$ is added to the hash table. Unfortunately, given that backbone links can have up to a million flows [5] today, this naive scheme would minimally require 64 Mbits of high speed memory[1]. Such large SRAM memory is expensive or not feasible for a modern router.

There are more efficient general-purpose algorithms for counting the number of distinct values in a multiset. In this paper we not only present a general-purpose counting algorithm – *multiresolution bitmap* – that has better accuracy than the best known prior algorithm, probabilistic counting [6], but introduce a whole family of counting algo-

[1]It must at least store the flow identifier, which in this example is 64 bits, for each of a million flows.

rithms that further improve performance by taking advantage of particularities of the specific counting application. Our *adaptive bitmap*, using the fact that the number of active flows doesn't change very rapidly, can count the number of distinct flows on a link that contains anywhere from 0 to 100 million flows with an average error of less than 1% using only 2 Kbytes of memory. Our *triggered bitmap*, optimized for running multiple concurrent instances of the counting problem, many of which have small counts, is suitable for detecting port scans and uses even less memory than running adaptive bitmap on each instance.

1.1 Problem Statement

A flow is defined by an *identifier* given by the values of certain header fields[2]. The problem we wish to solve is counting the number of distinct flow identifiers (flow IDs) seen in a specified *measurement interval*. For example, an intrusion detection system looking for port scans could count for each active source address the flows defined by destination IP and port and suspect any source IP that opens more than 3 flows in 12 seconds of performing a port scan. Other applications such as packet scheduling could prefer an alternate way of defining the number of active flows without using measurement interval: consider active the flows that have at least one packet in a queue that packets are added to and removed from dynamically. In this paper we mainly focus on the definition based on measurement intervals.

Also, while many applications define flows at the granularity of TCP connections, one may want to use other definitions. For example when detecting DoS attacks we may wish to count the number of distinct sources, not the number of TCP connections. Thus in this paper we use the term flow in this more generic way.

As we have seen, a naive solution using a hash table of flow IDs is accurate but takes too much memory. In high speed routers it is not only the cost of large, fast memories that is a problem but also their power consumption and the board space they take up on line cards. Thus, we seek solutions that use a very small amount of memory and have high accuracy. Usually there is a tradeoff between memory usage and accuracy, but we want to find algorithms where these tradeoffs are favorable. Also, since at high speeds the per packet processing time is very limited it is important that the algorithms use few memory accesses per packet. We describe algorithms that use only 1 or 2 memory accesses[3] and are simple enough to be implemented in hardware.

1.2 Motivation

Why is information about the number of flows useful? We describe five possible categories of use.

Detecting port scans: Intrusion detection systems warn of port scans when a source opens too many connections within a given time[4]. The widely deployed Snort intrusion detection system (IDS) [15] uses the naive approach of storing a record for each active connection. This is an obvious waste since most of the connections are not part of a port scan. Even for actual port scans, if the IDS only reports the number of connections we don't need to keep a record for each connection. Since the number of sources can be very high, it is desirable to find algorithms that count the number of connections of each source using little memory. Further, if an algorithm can distinguish quickly between suspected port scanners and normal traffic, the IDS need not perform expensive operations (e.g., logging) on most of the traffic, thus becoming more scalable in terms of memory usage and speed. This is particularly important in the context of the recent race to provide wire-speed intrusion detection [1].

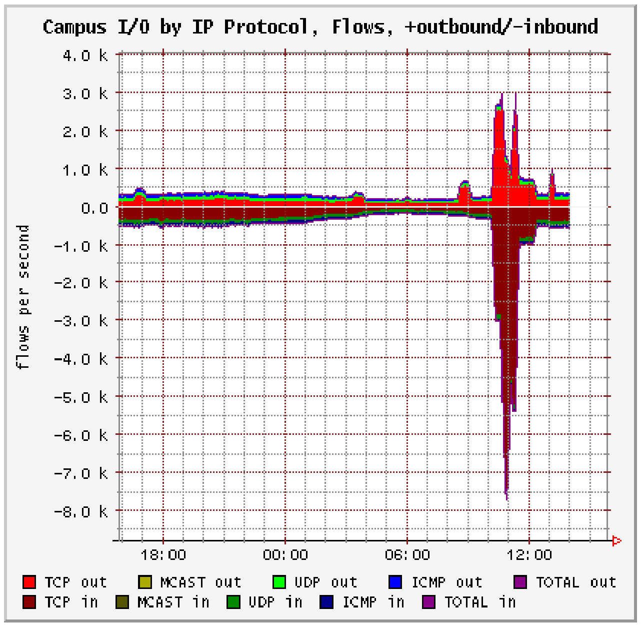

Figure 1: The flow count provided by Dave Plonka's FlowScan is used to detect denial of service attacks.

Detecting denial of service (DoS) attacks: FlowScan by David Plonka [14] is a popular tool for visualizing network traffic. It uses the number of active flows (see Figure 1) to detect ongoing denial of service attacks. While this works well at the edge of the network (i.e., the link between a large university campus and the rest of the Internet) it doesn't scale to the core. Also it relies on massive intermediate data (NetFlow) to compute compact results – could we obtain the useful information more directly? Mahajan et al. propose a mechanism that allows backbone routers to limit the effect of (distributed) DoS attacks [10]. While the mechanism assumes that these routers can detect an ongoing attack it does not give a concrete algorithm for it. Estan and Varghese present algorithms that can detect destination addresses or prefixes that receive large amounts of traffic [3]. While these can identify the victims of attacks it also gives many false positives because many destinations have large amounts of legitimate traffic. To differentiate between legitimate traffic and an attack we can use the fact that DoS tools use fake source addresses chosen at random[5]. If for

[2]We can also generalize by allowing the identifier to be a *function* of the header fields (e.g., using prefixes instead of addresses, based on routing tables).

[3]Actually, larger numbers of memory accesses are perfectly feasible at high speeds using SRAM and pipelining, but this increases the cost of the solution.

[4]While distributed port scans are possible, they are harder because the attacker has to control many endhosts it can scan from. If the number of hosts is not very large, each will have to probe many port-destination combinations thus running the risk of being detected.

[5]If the attack uses a small number of source addresses then

each suspected victim we count the number of sources of packets that come from some networks known to be sparsely populated, a large count is a strong indication that a DoS attack is in progress.

General measurement: Counting the number of active connections and the number of connections associated with each source and destination IP address is part of the CoralReef [9] traffic analysis suite. Other ways of counting the number of distinct values in given header fields can also provide useful data. One could measure the number of sources using a protocol version or variant to get an accurate image of protocol deployment. Alternatively, by counting the number of connections associated with each of the protocols generating significant traffic we can compute the average connection length for each protocol thus getting a better view of its behavior. Dimensioning the various caches in routers (packet classification caches, multicast route caches for source-group (S-G) state, and ARP caches) also benefits from prior measurements of typical workload.

Estimating the spreading rate of a worm: From Aug 1 to Aug 12 2001, while trying to track the Code Red worm [11], collecting packet headers for Code Red traffic on a /8 network produced 0.5 GB per hour of compressed data. In order to determine the rate at which the virus was spreading, it was necessary to count the number of distinct Code Red sources passing through the link. This was actually done using a large log and a hash table which was expensive in time and also inaccurate (because of losses in the log).

Packet scheduling: Many scheduling algorithms try to ensure that all flows can send at the current "fair share" of the available bandwidth. At high speeds it is not feasible to keep per-flow state. While there are scheduling algorithms that compute the fair share without using per-flow state (e.g., CSFQ [17], XCP [8]), they require explicit cooperation of edge routers or end hosts. Being able to count the number of distinct flows that have packets in the queue of the router might allow the router to estimate the "fair share" without outside help and could lead to scheduling algorithms that are less vulnerable to misbehaving end hosts or edge routers.

Thus, while counting the number of flows is usually insufficient by itself, it can provide a useful building block for complex tasks that range from detecting DoS attacks to fair packet scheduling.

2. RELATED WORK

The networking problem of counting the number of distinct flows has a well-studied equivalent in the database community: counting the number of distinct database records (or distinct values of an attribute). Thus, the major piece of related work is a seminal algorithm called *probabilistic counting*, due to Flajolet and Martin [6], introduced in the context of databases. We use probabilistic counting as a base to compare our algorithms against. Whang et al. address the same problem and propose an algorithm [18] that is equivalent to the simplest algorithm we describe (direct bitmap).

The insight behind probabilistic counting is to compute a metric of how uncommon a certain record is and keep track of the most uncommon record seen. If the algorithm sees very uncommon records, it concludes that the number of records is large. More precisely, for each record the algorithm computes a hash function that maps it to an L bit string (L is configurable). It then counts the number of consecutive zeroes starting from the least significant position of the hash result. If the algorithm sees records that hash to values ending in 0, 1 and 2 zeroes it concludes that the number of distinct records was $c2^2$ (c is a statistical correction factor), if it also sees hash values ending in 3 zeroes it estimates $c2^3$ and so on. This basic form can have an accuracy of at most 50% because possible estimates are a factor of 2 from each other. By dividing the hash values into *nmap* groups (*nmap* is configurable), and running a separate instance of the basic algorithm for each group and averaging over the estimates for the count provided by each of them, probabilistic counting reduces the error of its final estimate. We describe a family of algorithms that each outperforms probabilistic counting by an order of magnitude by exploiting application-specific characteristics.

In networking, there are general-purpose traffic measurement systems such as Cisco's NetFlow [12] or LFAP [13] that report per-flow records for very fine-grained flows. This is useful for traffic measurement. The information can be used to count flows (and this is what FlowScan [14] does), but is not optimized for such a purpose. Ideally, for high speed routers state should be in high speed SRAM (which is expensive and limited) to allow wire-speed forwarding. Because NetFlow state is so large, Cisco Routers write NetFlow state to slower DRAM which slows down packet processing. For high speeds sampling is used: only the sampled packets result in updates to the flow cache that keeps the per flow state. This affects the accuracy of the measurement data. Sampling works reasonably for estimating the traffic sent by large flows or large traffic aggregates, but has extremely poor accuracy for estimating the number of flows. This is because uniform sampling produces more samples of flows that send more traffic, thereby biasing any simple estimator that counts the number of flows in the sample and applies a correction.

Duffield et al. present two scalable methods for counting the number of active TCP flows based on samples of the traffic [2]. They rely on the fact that TCP turns the SYN flag on only for the packets starting a connection. The estimates are based on counts of the number of flows with SYN packets and the number of flows with non-SYN packets in the sampled data. While this is a good solution for TCP connections it cannot be applied to UDP or when we use a different definition for flows (e.g., when looking at protocol deployment statistics, we define a flow as all packets with the same source IP). Also counting flows in the sampled data can still be a memory-consuming operation that needs to be efficiently implemented.

The Snort [15] intrusion detection system (IDS) uses a memory-intensive approach similar to NetFlow to detect port scans: it maintains a record for each active connection and a connection counter for each source IP address. More elaborate algorithms have been used already in other settings. When controlling the medium access in wireless networks, some protocols rely on an estimate of the number of senders. The GRAP protocol [19] uses techniques equivalent to our direct bitmap and virtual bitmap to estimate this number¿ However GRAP has no equivalent of our more

it can be easily filtered out once those addresses are identified. Identifying those addresses can be done using previous techniques [3] because those few source addresses must send a lot of traffic each for the attack to be effective.

sophisticated multiresolution, adaptive, or triggered bitmap algorithms.

3. A FAMILY OF COUNTING ALGORITHMS

Our family of algorithms for estimating the number of active flows relies on updating a bitmap at run time. Different members of the family have different rules for updating the bitmap. At the end of the measurement interval (1 second, 1 minute, or even 1 hour), the bitmap is processed to yield an estimate for the number of active flows. Since we do not keep per-flow state, all of our results are estimates. However, we prove analytically and show through experiments on traces that our estimates are close to the actual values. The family contains three core algorithms and three derived algorithms. Even though the first two core algorithms (direct and virtual bitmap) were invented previously, we present them here because they form the basis of our new algorithms (multiresolution, adaptive, and triggered bitmaps), and because we present new applications in a networking context (as opposed to a database or wireless context).

We start in Section 3.1 with the first core algorithm, *direct bitmap*, that uses a large amount of memory. Next, in Section 3.2 we present the second core algorithm called *virtual bitmap* that uses sampling over the flow ID space to reduce the memory requirements. While virtual bitmap is extremely accurate, it needs to be tuned for a given anticipated range of the number of flows. We remove the "tuning" restriction of virtual bitmap with our third algorithm called *multiresolution bitmap*, described in Section 3.3, at the cost of increased memory usage. Finally, in Section 3.4 we describe the three derived algorithms. In this section we only describe the algorithms; we leave an analysis of the algorithms to Section 4.

3.1 Direct bitmap

The direct bitmap is a simple algorithm for estimating the number of flows. We use a hash function on the flow ID to map each flow to a bit of the bitmap. At the beginning of the measurement interval all bits are set to zero. Whenever a packet comes in, the bit its flow ID hashes to is set to 1. Note that all packets belonging to the same flow map to the same bit, so each flow turns on at most one bit irrespective of the number of packets it sends.

We could use the number of bits set as our estimate of the number of flows, but this is inaccurate because two or more flows can hash to the same bit. In Section 4.1, we derive a more accurate estimate that takes into account hash "collisions"[6]. Even with this better estimate, the algorithm becomes very inaccurate when the number of flows is much larger than the number of bits in the bitmap and the bitmap is almost full. The only way to preserve accuracy is to have a bitmap size that scales almost linearly with the number of flows, which is often impractical.

3.2 Virtual bitmap

The virtual bitmap algorithm reduces the memory usage by storing only a small portion of the big direct bitmap one would need for accurate results (see Figure 2) and extrapolating the number of bits set. This can also be thought of as sampling the flow ID space. The larger the number of flows the smaller the portion of the flow ID space we cover. Virtual bitmap generalizes direct bitmap: direct bitmap is a virtual bitmap which covers the entire flow ID space.

Unfortunately, a virtual bitmap does require tuning the "sampling factor" based on prior knowledge of the number of flows. If it differs significantly from what we configured the virtual bitmap for, the estimates are inaccurate. If the number of flows is too large the virtual bitmap fills up and has the same accuracy problems as an underdimensioned direct bitmap. If the number of flows is too small we have another problem: say the virtual bitmap covers 1% of the flow ID space and there are 50 active flows - if none of them hashes to the virtual bitmap, the algorithm will suppose the number of flows is 0, if 1 hashes, the algorithm will estimate 100, but it will never estimate 50. The optimal sampling factor obtains the best tradeoff between "collision errors" and "extrapolation errors".

While, in general, one wants an algorithm that is accurate over a wider range, we note that even an unadorned virtual bitmap is useful. For example, consider a security application where we wish to trigger an alarm when the number of flows crosses a threshold. The virtual bitmap can be tuned for this threshold and uses less memory than other algorithms that are accurate not just around the threshold, but over a wider range for the number of flows.

In Section 4 we derive formulae for the average error of the virtual bitmap estimates. The analysis also provides insight for choosing the right sampling factor. Perhaps surprisingly, the analysis also indicates that the average error depends only on the number of bits and not on the number of flows as long as the sampling factor is set to an optimal value. For example with 215 bytes the average error is 3%.

3.3 Multiresolution bitmap

The virtual bitmap is simple to implement, uses little memory, and gives very accurate results, but requires us to know in advance a reasonably narrow range for the number of flows. An immediate solution to this shortcoming is to use many virtual bitmaps, each using the same number of bits of memory, but different sampling factors, so that each is accurate for a different range of the number of active flows (different "resolutions"). The union of all these ranges is chosen to cover all possible values for the number of flows. When we compute our estimate, we use the virtual bitmap that is most accurate based on a simple rule that looks at the number of bits set. The "lowest resolution" bitmap is a direct bitmap that works well when there are very few flows. The "higher resolution" bitmaps cover a smaller and smaller portion of the flow ID space and work well when the number of flows is larger. The problem with the naive approach of using several virtual bitmaps of differing granularities is that instead of updating one bitmap for each packet, we need to update several, causing more memory accesses.

The main innovation in multiresolution bitmap is to maintain the advantages of multiple bitmaps configured for various ranges while performing a *single update* for each incoming packet. Figure 2 illustrates the direct bitmap, virtual bitmap, multiple bitmaps and multiresolution bitmap. Before explaining how the multiresolution bitmap works it can help to switch to another way of thinking about how the

[6]We assume in our analysis that the hash function distributes the flows randomly. In an adversarial setting, the attacker who knows the hash function could produce flow identifiers that produce excessive collisions thus evading detection. This is not possible if we use a random seed to our hash function.

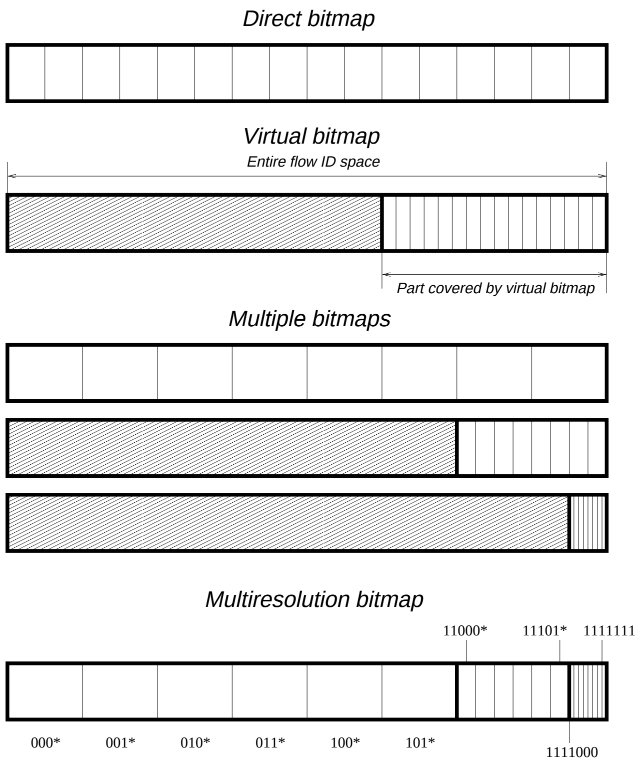

Figure 2: The multiresolution bitmap from this example uses a single 7-bit hash function to decide which bit to map a flow to. It gives results no less accurate than the 3 virtual bitmaps, thus covering a wide range for the number of flows, but it performs a single memory update per packet. Note that all the unfilled "tiles" from these bitmaps, despite their different sizes represent one bit of memory.

virtual bitmap operates. We can consider that instead of generating an integer, the hash function covers a continuous interval. The virtual bitmap covers a portion of this interval (the ratio of the sizes of the interval covered by the virtual bitmap and the entire interval is the sampling factor of the virtual bitmap). We divide the interval corresponding to the virtual bitmap into equal sized sub-intervals, each corresponding to a bit. A bit in the virtual bitmap is set to 1 if the hash of the incoming packet maps to the sub-interval corresponding to the bit. The multiple bitmaps solution is shown below the virtual bitmap solution in Figure 2.

A multiresolution bitmap is essentially a combination of multiple bitmaps of different "resolutions", such that a single hash is computed for each packet and only the highest resolution bitmap it maps to is updated. Thus each bitmap loses a portion of its bits which are covered by higher resolution bitmaps. But those bits can easily be recovered later (during the analysis phase) from the finer grained bitmaps by OR-ing together the bits in the higher resolution bitmaps that correspond to individual bits in the lower resolution bitmap. We call these regions with different resolutions components of the multiresolution bitmap. When we compute the estimate, based on the number of bits set in each component, we choose one of them as "base", estimate the number of flows hashing to it and all finer components and extrapolate.

In Section 4.3 we answer questions such as: how many bits should each component have, how many components do we need and what is the best ratio between the resolutions of neighboring components? In the technical report version of the paper [4] we show that multiresolution bitmaps are easy to implement even in hardware that can keep up with line speeds. Also, we compare our multiresolution bitmap to probabilistic counting showing that while both algorithms use nearly identical hashes to set bits, they interpret the data *very* differently, thus the differences in the accuracy of the results.

3.4 Derived algorithms

In this section we describe three derived algorithms for counting the number of active flows. *Adaptive bitmap*, described Section *3.4.1*, achieves both the accuracy of virtual bitmap and the robustness of multiresolution bitmap by combining them and relying on the stationarity of the number of flows. *Triggered bitmap* described in Section *3.4.2* combines direct bitmap and multiresolution bitmap to reduce the total amount of memory used by multiple instances of flow counting when most of the instances count few flows. In Section *3.4.3* we show how we can adapt the core algorithms to the alternate definition of active flows: the ones that have packets in a queue that supports arbitrary additions and removals (not those that send any packets during a fixed measurement interval).

3.4.1 Adaptive bitmap

It would be nice to have an algorithm that provides the best of both worlds: the accuracy of a well tuned virtual bitmap with the wide range of multiresolution bitmaps. Adaptive bitmap is such an algorithm that combines a large virtual bitmap and a small multiresolution bitmap. It relies on a simple observation: measurements show that the number of active flows does not change dramatically from one measurement interval to the other (so it is not suitable for tracking say attacks where sudden changes are expected). We use the small multiresolution bitmap to detect changes in the order of magnitude of the count, and the virtual bitmap for precise counting within the currently expected range. The number of flows we expect is the number of flows measured in the previous measurement interval. Assuming "quasi-stationarity", the algorithm is accurate most of the time because it uses the large, well-tuned virtual bitmap for estimating the number of flows. At startup and in the very unlikely case of dramatic changes in the number of active flows the multiresolution bitmap provides a less accurate estimate.

Updating these two bitmaps separately would require *two* memory updates per packet, but we can avoid the need for multiple updates by combining the two bitmaps into one. Specifically, we use a multiresolution bitmap in which r adjacent components are replaced by a single large component consisting of a virtual bitmap (where r is a configuration parameter). The location of the virtual bitmap within the multiresolution bitmap (i.e. which components it replaces) is determined by the current estimate of the count. If the current number of flows is small, we replace coarse components with the virtual bitmap. If the number of flows is large, we replace fine components with the virtual bitmap.

The update of the bitmap happens exactly as in the case of the multiresolution bitmap, except that the logic is changed slightly when the hash value maps to the virtual bitmap component.

3.4.2 *Triggered bitmap*

Consider the concrete example of detecting port scans. If one used a multiresolution bitmap per active source to count the number of connections, the multiresolution bitmap would need to be able to handle a large number of connections because port scans can use very many connections. The size of such a multiresolution bitmap can be quite large. However, most of the traffic is not port scans and most sources open only one or two connections. Thus using a large bitmap for each source is wasteful.

The triggered bitmap combines a very small direct bitmap with a large multiresolution bitmap. All sources are allocated a small direct bitmap. Once the number of bits set in the small direct bitmap exceeds a certain trigger value, a large multiresolution bitmap is allocated for that source and it is used for counting the connections from there on. Our estimate for the number of connections is the sum of the flows counted by the small direct bitmap and the multiresolution bitmap. This way we have accurate results for all sources but only pay the cost of a large multiresolution bitmap for the sources that open many connections.

As described so far, this algorithm introduces a subtle error that makes a small change necessary. If a flow is active both before and after the large multiresolution bitmap is allocated it gets counted by both the direct bitmap and the multiresolution bitmap. Only using the multiresolution bitmap for our final estimate is not a solution either because than we would not count the flows that were active only before the multiresolution bitmap was allocated. To avoid this problem we change the algorithm the following way: after the multiresolution bitmap is allocated, we only map to it those flows that do not map to one of the bits already set in the direct bitmap. This way if the flows that set the bits in the direct bitmap send more packets, they will not influence the multiresolution bitmap. It's true that the multiresolution bitmap doesn't catch all the new flows, just the ones that map to one of the bits not set in the direct bitmap. This is equivalent to the "sampling factor" of the virtual bitmap and we can compensate for it (see Section 4.1).

3.4.3 *Handling packet removals*

We mentioned earlier that counting the the number of flows that have packets in the queue of a router can help determine the "fair share" used by the scheduling algorithm. In this case, we need to not only handle the case of new packets arriving but also the case of packets getting removed. Direct bitmap, virtual bitmap and multiresolution bitmap can be easily modified to handle this case by replacing every bit with a counter. The width of the counters is given by the maximum number of packets the queue can accommodate (which also puts a limit on the number of distinct flows that can have packets in the queue). When the queue is empty all counters are 0. When a new packet arrives, the counter it maps to is incremented. When a packet is removed from the queue, the counter is decremented. We use the number of counters with value zero to compute our estimate of the number of active flows exactly the same way we use the number of zero bits to estimate the number of active flows in a measurement interval. A counter will be zero if and only if no active flows map to it.

4. ALGORITHM ANALYSIS

In this section we provide the analyses of the statistical behavior of the bitmaps used by our algorithms. We focus on three types of results. In Section 4.1, we derive formulae for estimating the number of active flows based on the observed bitmaps. In Section 4.2, we analytically characterize the accuracy of the algorithms by deriving formulae for the average error of the estimates. In Section 4.3, we use the analysis to derive rules for dimensioning the various bitmaps so that we achieve the desired accuracy over the desired range for the number of flows.

4.1 Estimate Formulae

Direct bitmap: To derive a formula for estimating the number of active flows for a direct bitmap we have to take into account collisions. Let b be the size of the bitmap. The probability that a given flow hashes to a given bit is $p = 1/b$. Assuming that n is the number of active flows, the probability that no flow hashes to a given bit is $p_z = (1-p)^n \approx (1/e)^{n/b}$. By linearity of expectation this formula gives us the expected number of bits not set at the end of the measurement interval $E[z] = bp_z \approx b(1/e)^{n/b}$. If the number of zero bits is z, Equation 1 gives our estimate $\widehat{n}$ for the number of active flows. Whang et al. also show that this is the maximum likelihood estimator for the number of active flows [18].

$$\widehat{n} = b\ \ln\left(\frac{b}{z}\right) \tag{1}$$

Virtual bitmap: Let α be the "sampling factor" (the ratio of the sizes of the interval covered by the virtual bitmap b and the entire hash space h). The probability for a given flow to hash to the virtual bitmap is equal to the sampling factor $p_v = \alpha = b/h$. Let m be the number of flows that actually hash to the virtual bitmap. Its probability distribution is binomial with an expected value of $E[m] = \alpha n$. We can use Equation 1 to estimate m and based on that we obtain Equation 2 for the estimate of the number of active flows n.

$$\widehat{n} = \frac{1}{\alpha}\ b\ln\left(\frac{b}{z}\right) = h\ln\left(\frac{b}{z}\right) \tag{2}$$

Multiresolution bitmap: The multiresolution bitmap is a combination of many components, each tuned to provide accurate estimates over a particular range. When we compute our estimate we don't know in advance which component is the one that provides the most accurate estimate (we call this the base component). As we will see in Section 4.2, we obtain the smallest error by choosing as the base component the coarsest component that has no more than set_{max} bits (lines 1 to 5 in Figure 3). set_{max} is a precomputed threshold based on the analysis from Section 4.2. Once we have the base component, we estimate the number of flows hashing to the base and all the higher resolution ones using Equation 1 and add them together (lines 13 to 17 in Figure 3). To obtain the result we only need to perform the multiplication corresponding to the sampling factor (lines 18 and 19). Other parameters used by this algorithm are the

```
ESTIMATEFLOWCOUNT
1   $base = c - 1$
2   while $base > 0$ and $bitsSet(component[base]) \leq set_{max}$
3       $base = base - 1$
4   endwhile
5   $base = base + 1$
6   if $base == c$ and $bitsSet(component[c]) > setlast_{max}$)
7       if $bitsSet(component[c]) == b_{last}$
8           return "Cannot give estimate"
9       else
10          warning "Estimate might be inaccurate"
11      endif
12  endif
13  $m = 0$
14  for $i = base$ to $c - 1$
15      $m = m + b \ln(b/bitsZero(component[i]))$
16  endfor
17  $m = m + b_{last} \ln(b_{last}/bitsZero(component[c]))$
18  $factor = k^{base-1}$
19  return $factor * m$
```

Figure 3: Algorithm for computing the estimate of the number of active flows for a multiresolution bitmap. We first pick the base component that gives the best accuracy then add together the estimates for the number of flows hashing to it and all higher resolution components and finally extrapolate.

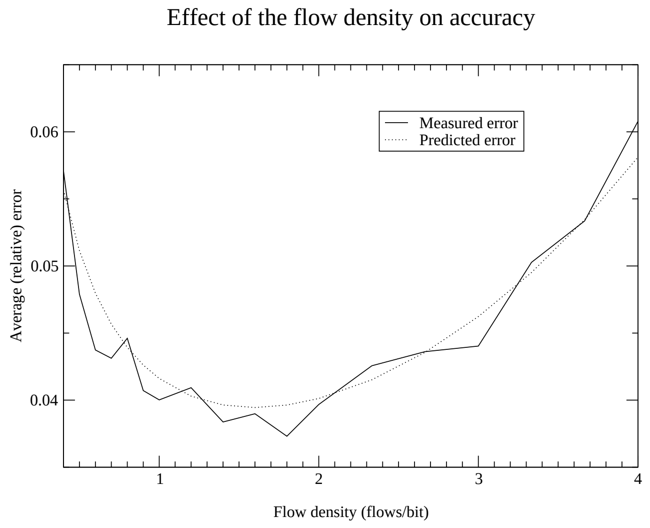

Figure 4: When the flow density is too low, the "sampling error" takes over, when it is too high "collision error" is the main factor. We get the best accuracy for a flow density of around $\rho = 1.6$. The estimate from Equation 4 matches well the experimental results being slightly conservative (larger). See Section 5.1 for details on the experiment that produced this result.

ratio k between the resolutions of neighboring components and b_{last} the number of bits in the last component (which is different from b).

Adaptive bitmap: The algorithm for adaptive bitmap is very similar to multiresolution bitmap. The main difference is that we use different threshold for selecting the big component as base. For brevity, we omit the algorithm.

Triggered bitmap: If the triggered bitmap did not allocate a multiresolution bitmap, we simply use the formula for direct bitmaps (Equation 1). Let's use g for the number of bits that have to be set in the direct bitmap before the multiresolution bitmap is allocated and d for the total number of bits in the direct bitmap. If the multiresolution bitmap is deployed, we use the algorithm from Figure 3 to compute the number of flows hashing to the multiresolution bitmap, multiply that by $d/(d - g)$ and add the estimate of the direct bitmap.

4.2 Accuracy

To determine the accuracy of these algorithms we look at the standard error of our estimate $\widehat{n}$, that is the standard deviation of the ratio $\widehat{n}/n$. We also refer to this quantity as the average (relative) error $SD[\widehat{n}/n] = SD[\widehat{n}]/n$. One parameter that is useful in these analyses is the flow density ρ defined as the average number of flows that hash to a bit.

Direct bitmap: While our formula for estimating the number of active flows accounts for the expected collisions it doesn't always give exact results because the number of collisions is random. Equation 3 approximates the average error of a direct bitmap based on the Taylor expansion of Equation 1 as derived by Whang et al. [18]. The result is not exact because because less significant terms of the Taylor expansion were omitted. Whang et al. also show that the approximation does not lead to serious inaccuracies for configurations one expects to see in practice. They also show that the distribution of the number of bits set is asymptotically normal so errors much larger than the standard error are very unlikely [18]. For example, for a direct bitmap configured to operate at an average error of 10% for flow densities up to 2, the value of the average error we get by including the next term of the Taylor series is only 2% away from the approximation (i.e., the actual average error can be at most 10.2% instead of 10%). The inaccuracy introduced by the approximation decreases further as the number of bits increases.

$$SD\left[\frac{\widehat{n}}{n}\right] \approx \frac{\sqrt{e^{\rho} - \rho - 1}}{\rho\sqrt{b}} \qquad (3)$$

Virtual bitmap: Besides the randomness in the collisions, there is another source of error for the virtual bitmap: we assume that the ratio between the number of flows that hash to the physical bitmap and all flows is exactly the sampling factor while due to the randomness of the process the number can differ. In Appendix A we analyze these two errors and how they interact. Equation 4 takes into account their cumulative effect on the result. When the flow density is too large the error increases exponentially because of the collision errors. When it is too small, the error increases as the sampling errors take over. Our analysis also shows that the terms ignored by the approximations do not contribute significantly and that the bound is tight. Figure 4 presents a typical result comparing the measured average error from simulations on traces of actual traffic to the value from Equation 4.

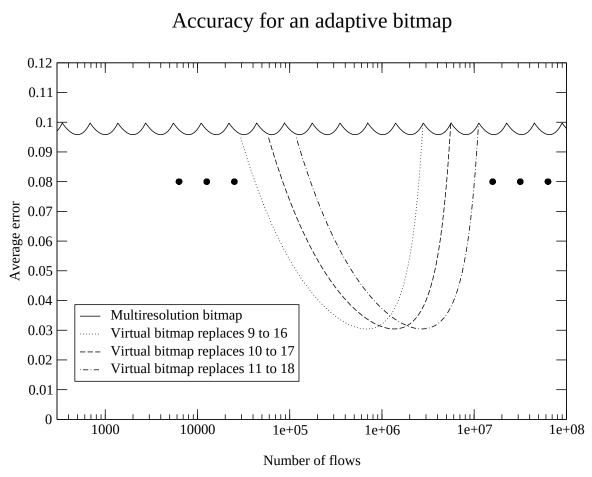

Figure 5: The large virtual bitmap replaces 6 of the components of the multiresolution bitmap. The size of the normal components is $b = 64$ bits and the size of the large virtual bitmap is $v = 1627$ bits. The adaptive bitmap guarantees an average error of at most 10% over the whole range, but if the number of flows falls into the "sweet spot" the average error can be as low as 3.1%

$$SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{e^{\rho}-1}}{\rho\sqrt{b}} \tag{4}$$

Multiresolution bitmap: To compute the average error of the estimate of the multiresolution bitmap, we should take into account separately the collision errors of all components finer than the base. This would result for a different formula for each component that would be used as base. Equation 5 is a slightly weaker bound that holds for all components but the last one as long as the number of bits in the last component b_{last} is large enough. The details of its derivation can be found in Appendix A. Equation 5 bounds quite tightly the average error for a normal component. For the last component of the multiresolution bitmap we use Equation 4 directly.

$$SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{\frac{k-1}{k}\left(e^{\rho}+e^{\rho/k}-2\right)+e^{\rho/k^2}-1}}{\rho\sqrt{\frac{bk}{k-1}}} \tag{5}$$

Adaptive bitmap: The error of the estimates of the adaptive bitmap depends strongly on the number of flows: the errors are much larger if the number of flows is unexpectedly large or small. The exact formulas, omitted for brevity are not very different from the ones seen so far. We give an example instead. Figure 5 gives the average error as predicted by our formulae for the adaptive bitmap we use in for measurements (Section 5.3). We first represent the average error of the original multiresolution bitmap and then the average error we obtain by replacing various groups of 8 consecutive components with the virtual bitmap. It is apparent from this figure that by changing which components are replaced by the virtual bitmap we can change the range for which the adaptive bitmap is accurate.

Algorithm	Memory (bits)
Direct bitmap	$< N/\ln(N\epsilon^2+1)$
Virtual bitmap	$1.54413865/\epsilon^2$
Multiresolution bmp.	$0.9186\ln(N\epsilon^2)/\epsilon^2 + ct.$
Adaptive bitmap	$\gtrapprox 1.54413865/\epsilon^2$

Table 1: The size of the direct bitmap scales sublinearly with N but worse than $N/\ln(N\epsilon^2+1)$, the size for the virtual bitmap is proportional to the inverse of the square of the average error, the size of the multiresolution bitmap scales with the logarithm of the number of flows over the square of the average error and the adaptive bitmap delivers under certain assumptions the accuracy of the virtual bitmap by adapting dynamically to the number of active flows.

4.3 Configuring the bitmaps

In this section we address the configuration details and implicitly the memory needs of the bitmap algorithms. All measurement results are in Section 5. The two main parameters we use to configure the bitmaps are the maximum number of flows one wants them to count N and the acceptable average *relative* error ϵ. We base our computations on the formulas of the previous section.

Direct bitmap: If we would keep $\rho = N/b$ constant as N increased ϵ would improve proportionally to $1/\sqrt{N}$ (which is proportional to $1/\sqrt{b}$). So as N increases the flow density that gives us the desired accuracy also increases. Therefore by ignoring the constant term under the square root in Equation 3 we get a tight bound on how b scales. $\epsilon^2 \lessapprox (e^{\rho}-\rho)/(\rho^2 b)$ so $\epsilon^2 N + 1 \lessapprox e^{\rho}/\rho < e^{\rho}$. From here $\rho > \ln(\epsilon^2 N+1)$ and thus $b < N/\ln(\epsilon^2 N+1)$. We claim that for large values of N while this closed form bound is not tight it is not very far off either. For example for $N = 1,000,000$ and $\epsilon = 10\%$ the bound gives 108,572 bits while the actual value is 85,711 bits. Of course, for configuring a direct bitmap we recommend solving Equation 3 numerically for b (with ρ replaced by N/b).

Virtual bitmap: The average error of the virtual bitmap given by Equation 4 is minimized by a certain value of the flow density. Solving numerically we get $\rho_{optimal} = 1.593624$ and this corresponds to around 20.3% of the bits of the bitmap being not set. By substituting, we obtain the average error for this "sweet spot" flow density $\epsilon \lessapprox 1.242633756330/\sqrt{b}$. By inverting this we obtain the formula from Table 1 for the number of bits of physical memory we need to achieve a certain accuracy. When we need to configure the virtual bitmap as a trigger, we set the sampling factor such that at the threshold the flow density is 1.593624. For this application, if we have 155 bits, the average error of our estimate is at most 10% no matter how large the threshold. If we have 1,716, the average error is at most 3%, and if we have 15,442 it is at most 1%. If we want to have at most a certain error for a range of flow counts between N_{min} and N_{max}, we need to solve the problem numerically by finding a $\rho_{min} < \rho_{optimal}$ and a $\rho_{max} > \rho_{optimal}$ so that $\rho_{max}/\rho_{min} = N_{max}/N_{min}$ and ρ_{min} and ρ_{max} produce the same error. Once we have these values, we can compute the sampling factor for the virtual bitmap and the number of bits.

Multiresolution bitmap: For the multiresolution bitmap, we have to ensure that the average error doesn't exceed the

k	ρ_{min}	ρ_{max}	coefficient $f(k)$	$f(k)/\ln(k)$
2	1.3372	2.6744	0.6367	0.9186
3	0.9750	2.9250	1.0318	0.9392
4	0.7856	3.1426	1.3470	0.9717

Table 2: The operating range of the components of the multiresolution bitmap is between ρ_{min} and ρ_{max}. The coefficient and the desired accuracy determine the size of the components $b = f(k)/\epsilon^2$. The larger the ratio between the resolutions of neighboring components k, the wider the range covered by a single component and the larger the component.

desired value over the whole range from 0 to N. We divide the range among components. Configuring a component is very much like configuring a virtual bitmap for a range, except we use Equation 5. We find two flow densities ρ_{min} and ρ_{max} that give the same error under the constraint that $\rho_{max}/\rho_{min} = k$ (k is the ratio between the resolutions of neighboring components). We choose the bitmap size b for the normal components (all except the last one) such that at ρ_{min} and ρ_{max} we get the desired accuracy $b = f(k)/\epsilon^2$ where the coefficient $f(k)$ depends on k. Table 2 contains the values of ρ_{min}, ρ_{max} and the coefficient used for determining the bitmap size for three useful values for k. The base component is the one with a flow density between ρ_{min} and ρ_{max}, so the threshold used by the algorithm (Figure 3) to select the base component is $set_{max} = b(1 - e^{-\rho_{max}})$.

We can choose the number of components such that the last normal component (the penultimate overall) covers the end of the range N: $c = 2 + \lceil log_k(N/(\rho_{max} b))\rceil$. The total size of the multiresolution bitmap is $Mem = b*(c-1)+b_{last}$, thus ignoring the additive constants, the asymptotic memory usage is $Mem \approx \ln(N\epsilon^2)/\epsilon^2 f(k)/\ln(k)$. By allocating more bits to the last component than what it needs in order to make the penultimate component accurate at ρ_{max}, it can also provide accurate enough estimates and this allows us to reduce the number of components in the bitmap. The algorithm for computing the optimal configuration[7] is long but not very complicated: it evaluates some choices for b_{last} and c and picks the best one.

The ratio $f(k)/\ln(k)$ gives the asymptotic memory usage for a certain choice of k and we can see from Table 2 that $k = 2$ is the best choice[8]. The algorithm is very easy to implement in hardware if k, $bk/(k-1)$ and b_{last} are powers of two. Under these constraints, sometimes the choice of $k = 4$ gives a smaller memory usage because the size b of the components it needs to achieve the desired average error ϵ "fits better" the powers of two. Therefore when configuring the algorithm for a hardware implementation that has these limitations it is best to check both values of $k = 2$ and $k = 4$. [9]

Adaptive bitmap: For brevity we omit the detailed discussion of the configuration of the adaptive bitmap. In

[7]The full algorithm is presented in the technical report [4].
[8]There are some very rare cases when $k = 3$ gives a slightly smaller memory usage. This is because the number of components cannot be fractional and the components for $k = 3$ "fit better" to the given N and ϵ.
[9]We found no set of parameters N,ϵ for which $k = 8$ worked better than both $k = 2$ and $k = 4$

r	v/b	improvement
2	2.3626	1.1725
3	4.4861	1.4488
4	8.0603	1.8468
5	14.3252	2.4029
6	25.5510	3.1709
7	45.9411	4.2265
8	83.3330	5.6754
9	152.4217	7.6641
10	280.8654	10.3959
11	520.9068	14.1524
12	971.5300	19.3240

Table 3: As we increase the number of components r replaced by the virtual bitmap, the size of the virtual bitmap v almost doubles for each new component replaced. The ratio between the average error of the large virtual bitmap and the multiresolution bitmap also increases exponentially, but at a slower rate than the size of the virtual bitmap.

Table 3 we report the costs and benefits of the adaptive bitmap. The first column lists the number r of normal components we replace with the large one. The next column lists the number of bits the large component needs to have (compared to the number of bits of a normal component) to ensure that the adaptive bitmap never has a worse average error than the original multiresolution bitmap. The third column lists the ratio between the average error of multiresolution bitmap and the "sweet spot" average error of the adaptive bitmap. The memory usage reported in Table 1 is derived based on the observation that most of the memory of the adaptive bitmap is used by the "virtual bitmap" component.

5. MEASUREMENT RESULTS

We group our measurements into 4 sections corresponding to the 4 important algorithms presented: virtual bitmap, multiresolution bitmap, adaptive bitmap and triggered bitmap. Part of the measurements are geared toward checking the correctness of the predictions of our theoretical analysis and part are geared toward comparing the performance of our algorithms with probabilistic counting or other existing solutions.

For our experiments, we used 3 packet traces, an unencrypted one from CAIDA captured on the 6th of August 2001 on an OC-48 backbone link and two encrypted traces from the MOAT project of NLANR captured on the 11th of November 2002 on the connection points of two university campuses to the Internet. The unencrypted trace is very long; for some experiments we also used a 90 second slice of the unencrypted trace as a fourth trace. We usually set the measurement interval to 5 seconds. We chose 5 seconds because it appears to be a plausible interval for looking at the number of active flows: it is larger than the round-trip times we can expect in the Internet and it is above the rate a slow modem link sends packets. In all experiments we defined the flows by the 5-tuple of source and destination IP addresses, ports, and protocol. Table 4 gives a summary description of the traces we used. All algorithms used equivalent CRC-based hash functions with random generator functions.

Name	No. of flows (min/avg/max)	Length (s)	Encr.
MAG+	93,437 / 98,424 / 105,814	4515	no
MAG	99,264 / 100,105 / 101,038	90	no
COS	17,716 / 18,070 / 18,537	90	yes
IND	1,964 / 2,164 / 2,349	90	yes

Table 4: The traces used for our measurements

5.1 Virtual bitmap

We performed experiments to check the validity of Equation 3 for various configurations on many traces. Figure 4 shows a typical result. More results can be found in the technical report version of the paper [4]. Our measurements confirm that Equation 3 gives a tight and slightly conservative bound on the average error (conservative in the sense that actual errors are usually somewhat smaller than predicted by the formula). The results also confirm that we get the best average error for a virtual bitmap of a given size when the flow density is around $\rho = 1.6$.

We also compare the average error of the virtual bitmap to probabilistic counting using the same amount of memory for a variety of configurations and traces. Because our major contributions are the remaining schemes, we provide here only one sample result. For the COS Trace, using 1,716 bits our analysis predicts an expected error 3%. Over 20 runs, for the 18 measurement intervals, the actual average error (computed as square root of the average of squares) for virtual bitmap is only 2.773% with a maximum of 9.467%. This is not just a further confirmation that Equation 3 gives a tight bound on the average error, but it also shows that errors much larger than the average error are very unlikely. On the other hand, probabilistic counting configured to handle up to 100,000 flows had an average error of 6.731% with a maximum of 27.336%. While this is an unfair comparison in general (virtual bitmap requires knowing in advance the range of final count values), it does fairly indicate our major message: a problem-specific counting method for a specific problem like threshold detection can significantly outperform a one-size-fits-all technique like probabilistic counting.

5.2 Multiresolution bitmap

This set of experiments compares the average error of the multiresolution bitmap and probabilistic counting. A meaningful comparison is possible if we compare the two algorithms over the whole range for the number of flows. Since our traces have a pretty constant number of flows, we use a synthetic trace for this experiment. We used the actual packet headers from the MAG+ trace to generate a trace that has a different number of flows in each measurement interval: from 10 to 1,000,000 in increments of 10% with a jitter of 1% added to avoid any possible effects of "synchronizations" with certain series of numbers.

We ran experiments with multiresolution bitmaps tuned to give an average error of 1%, 3% and 10% for up to 1,000,000 flows and probabilistic counting configured for the same range with the same amount of memory. We had 500 runs for each configuration of both algorithms with different hash functions.

Figures 6 to 8 show the results of the experiments. We can see that in all three experiments, the average error of

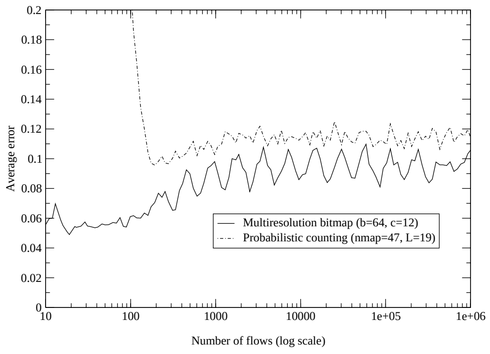

Figure 6: Configured for an average error of 10%

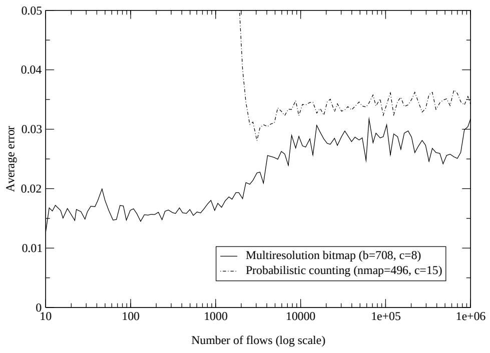

Figure 7: Configured for an average error of 3%

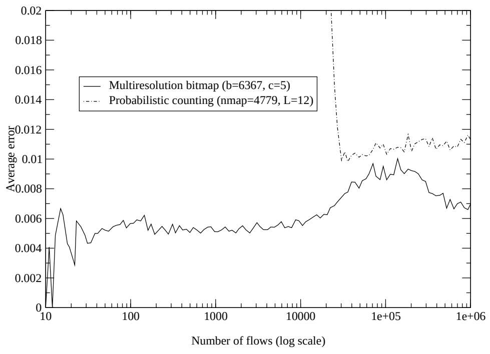

Figure 8: Configured for an average error of 1%

the multiresolution bitmap is better than predicted for small values, because we have no "sampling error" when the number of flows is small. We explain the periodic "fluctuations" of average error from figure 6 by occasional incorrect choice of the base component. The peaks correspond to where components are least accurate and hand off to each other. The peaks are more pronounced in this figure than the others because due to the small number of bits in each compo-

Trace	Adaptive bitmap (min/avg/max)	Probabilistic counting (min/avg/max)
MAG+	-4.402/1.066/4.717%	-9.525/2.820/13.262%
COS	-1.879/0.748/1.950%	-6.946/2.759/7.621%
IND	-1.767/0.601/1.772%	2.400/10.214/17.724%

Table 5: Comparison of adaptive bitmap and probabilistic counting, each using 16Kbits of memory

nent, it happens more often that not the best component is used as a base for the estimation. In Figure 7 and especially in Figure 8 there is a visible decrease in the error for the multiresolution bitmap when the number of flows approaches the upper limit. The reason is that the last component is much larger than the normal ones and provides more accurate results.

Probabilistic counting is worse than the multiresolution bitmap, especially for small values. We show in the technical report version of the paper [4] that the data collected by the two algorithms is equivalent, so it might be surprising that their accuracies are so different. We attribute the large errors of probabilistic counting for low values to the way it evaluates the collected data. The ability of multiresolution bitmap to be accurate on the low end of the range too can lead to simpler, more robust systems. We attribute the worse error of probabilistic counting for higher values mostly to the suboptimal dimensioning of the algorithm (as recommended in [6]).

5.3 Adaptive bitmap

The experiments from this section compare adaptive bitmap and probabilistic counting on all three traces. The results are presented in Table 5. All of the algorithms were configured to use 16 Kbits of memory. For each algorithm we report the largest errors in both directions and the average error based on 20 runs with different hash functions.

The algorithms were configured to give the best possible average error and work up to 100,000,000 flows. For the adaptive bitmap we used as a base a multiresolution bitmap with an average error of 10% with $k = 2$, $b = 64$, $c = 19$ and $b_{last} = 169$. The virtual bitmap component is $15,063$ bits large and replaces 9 components of the multiresolution bitmap. For the adaptive bitmap we did not include in our computations the first measurement interval when the adaptive bitmap was not tuned to the traffic. For the probabilistic counting we used $nmap = 744$ bitmaps of $L = 22$ bits each. Adaptive bitmap is roughly 3 times more accurate than probabilistic counting. For the IND trace which has a very small number of active flows probabilistic counting has very bad error and is actually biased towards overestimating. This is the same as the problem we noticed in the previous section. The major message here is that an adaptive bitmap can achieve almost the same benefits of virtual bitmap (e.g., order of magnitude reduction in memory for same accuracy) when the number of flows does not vary dramatically, as seems common in many networking applications.

5.4 Triggered bitmap

So far, all our measurements have focused on one instance of the counting problem to be used as a building block for solving more complex problems. The experiments from this section give a better image of how using our algorithms can affect the resource consumption of an entire system.

Measurement interval	Snort	Prob. count.	Triggered bmp.
12 sec	1,968K	2,474K	381K
600 sec	50,791K	22,876K	5,725K

Table 6: The memory usage of port scan detection algorithms (Kbytes)

We first address port scan detection that uses a large number (one per source) of instances of the counting problem multiplying the impact of any memory our algorithm can save. We use a definition of a port scan equivalent to the definition in the default Snort configuration: a source is flagged as a port scanner if it has at least 4 connections in a 12 second measurement interval. In the second experiment we extend the measurement interval to 10 minutes to evaluate the algorithms against this more demanding definition. We ignore many of the details of the operation of Snort (e.g., reliance on TCP flags to classify connections) and concentrate on the core task of counting connections.

For the triggered bitmap we chose a configuration that is convenient to implement on a 32-bit machine: a direct bitmap of 4 bytes and a multiresolution bitmap with 11 components of 4 bytes each (except the last one which is 8 bytes). The multiresolution bitmap is allocated after 8 bits are set in the direct bitmap. By our analysis the multiresolution bitmap should ensure an average error of at most 14.1% for up to 43,817 connections and at most 15.5% for up to 175,269 connections.

We compute memory usage of Snort based on the number of sources and connections active during the measurement interval. What we actually use is a not an accurate model of the actual memory usage of Snort (which uses inefficient structures such as multiple linked lists) but the minimum that any implementation using the naive algorithm would have to allocate: 8 bytes for the IP address and a counter for each source and 9 bytes (destination IP, source port, destination port, type) for the identifier of each active connection. We also compute the memory usage of a solution directly applying probabilistic counting with a configuration similar to our multiresolution bitmap (48 bytes for the algorithm + 4 bytes for the IP address for each source). Our triggered bitmap algorithm consumes 8 bytes for each active source (the IP address + the direct bitmap) plus the additional 48 bytes for the sources that trigger the allocation of the multiresolution bitmap.

We used two configurations, one with a 12 second prefix and one with a 600 second prefix of the MAG+ trace. For each configuration we had 20 runs of with the triggered bitmap algorithm, using different random hash functions. The average of the error for flows that had at least 4 connections was 13.6%.[10] Our algorithm reported 84.6% of the

[10]This is an average over all sources. We noticed some "peculiarities": for sources that had 4 connections the average error was around 10.5% , for those with 5 around 11%, for those with 6 it was 18%, for those with 8 around 11.5% while for all others the averages were roughly in the range 14%-15.5%. We explain these as effects of having such a small

sources with 4 connections as reaching the threshold, 98.1% of those with 5, and all (100%) of the sources that had at least 8 connections. In Table 6 we report the *maximum* of triggered bitmap over the 20 runs. Triggered bitmap uses roughly 5 times less memory than snort with the first configuration. For the more ambitious second configuration the gain increases to a factor of 9. With both configurations triggered bitmap used less memory than probabilistic counting.

What do these results mean to a security analyst? Snort, of course, uses the classical measure of detecting n connections with a maximum inter-event spacing of t. By default, Snort uses values such as n=4, t=3. Our technique uses significantly less memory at the expense of possibly missing port scanners. However, the probability of a port scanner not being detected decreases exponentially with the number of connections it opens. For example, the probability is 1.87% at 5 connections, 0.23% at 6, 0.03% at 7, etc. Using Snort's timing requirements, a fifth event must arrive within $t = 3$ seconds of the fourth event if the scan continues. Thus, we detect a continuing scan with probability 98.13% within 3 seconds and 99.77% within 6 seconds. Note also that port scans are usually the result of a brute-force network exploration such as Nmap [7] or Code Red [11]. Such tools frequently touch not just a handful of addresses, but an entire block of contiguous addresses. Thus, it is reasonable to expect a scan to continue after 4 events.

There are preliminary results on the use of triggered bitmap in a part of the CoralReef traffic analysis suite [9] computing per-IP source and destination statistics. The implementation and measurements reported here are the work of Ken Keys with contribution from David Moore, both from Caida. On a 10 minute OC-48 trace, the original application uses 316 megabytes of main memory. The improved version used a triggered bitmap with a 128 bit direct bitmap that allocated a multiresolution bitmap configured for an error of 5% after 4 bits were set. The memory usage decreased to 44 megabytes while the average error was 4.41%. The average running time was 349 seconds which is 29% below the running time of the original application (491 seconds).

Finally, note that because our algorithms reduce the memory usage by as much as an order of magnitude, they also enable detection of stealthy slow scans using the same amount of memory that naive algorithms use for fast scans. Because the memory required for each source is greatly reduced with our algorithms, we can afford to count more sources at a time. As a result, we can avoid timing-out state as aggressively as Snort and keep counting sources with longer inter-arrival times between events detecting more stealthy port scans, a goal of many detection systems [16].

6. CONCLUSIONS

Using a suitably general definition of a flow, counting the number of active flows is at the core of a wide variety of security and networking applications such as detecting port scans and denial of service attacks, tracking virus infections, calibrating caching, etc. In this paper we provide a family of bitmap algorithms solving the flow counting problem using extremely small amounts of memory. Most of the algorithms can be implemented at wire speeds (8 nsec per packet for OC-768) using SRAM since they access at most one memory location per packet, and can be implemented using simple hardware (CRC based hash functions, multipliers, and multiplexers). With the exception of direct and virtual bitmap, the algorithms are introduced for the first time in this paper.

direct bitmap.

The best known algorithm for counting distinct values is probabilistic counting. Our algorithms need less memory to produce results of the same accuracy. This can translate into savings of scarce, fast memory (SRAM) for hardware implementations. It can also help systems that use cheaper DRAM to scale to larger instances of the problem.

In comparing head-on with probabilistic counting, our multiresolution algorithm works under the same assumptions and provides an error orders of magnitude lower when the number of flows is small and is slightly better for higher values. However, we believe our biggest contribution is as follows. By exposing the simple building blocks and analysis behind multiresolution counting, we have provided a family of *customizable* counting algorithms (Table 7) that application and hardware designers can use to reduce memory even further by exploiting application characteristics.

Thus, virtual bitmap is well-suited for triggers such as detecting DoS attacks, and uses 215 bytes to achieve an error of 2.773% compared to 2,076 bytes for probabilistic counting. Adaptive bitmap is suited to flow measurement applications and exploits stationarity to require 8 times less memory than probabilistic counting on sample traces. Triggered bitmap is suited to running multiple instances of counting where many instances have small count values (e.g., port scanning) requiring only 5.6 Mbytes on a 10 minute trace compared to the 49.6 Mbytes required by the naive algorithm and 22.3 Mbytes required by probabilistic counting. Using triggered bitmap resulted in a reduction by 29% in the running time and a factor of seven in the total memory usage of a traffic analysis application from the CoralReef suite. Given that low-memory counting appears to be useful in applications beyond networking which have different characteristics, we hope that the base algorithms in this paper will be combined in other interesting ways in architecture, operating systems, and even databases.

7. ACKNOWLEDGEMENTS

We thank Vern Paxson, David Moore, Philippe Flajolet, Marianne Durand, Alex Snoeren and K. Claffy, Stefan Savage and Florin Baboescu for extremely valuable conversations. This work was made possible by NSF Grant ANI-0137102 and the Sensilla project sponsored by NIST Grant 60NANB1D0118.

8. REFERENCES

[1] Cisco offers wire-speed intrusion detection, December 2000. http://www.nwfusion.com/ reviews/ 2000/ 1218rev2.html.

[2] Nick Duffield, Carsten Lund, and Mikkel Thorup. Properties and prediction of flow statistics from sampled packet streams. In *SIGCOMM Internet Measurement Workshop*, November 2002.

[3] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM*, August 2002.

[4] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. Technical Report 0738, CSE Department, UCSD, March 2003.

Setting	Algorithm	Application
General counting	Multiresolution bitmap	Tracking virus infections
Accuracy important only over a narrow range	Virtual bitmap	Triggers (e.g. for detecting DoS attacks)
Count is probably in a narrow range (stationarity)	Adaptive bitmap	Measurement
Small memory usage as long as count is small	Triggered bitmap	Detecting port scans
Flows dynamically added and deleted	Increment-decrement algorithms	Scheduling

Table 7: The family of bitmap counting algorithms: each algorithm is best suited for a different setting.

[5] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM*, December 1999.

[6] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, October 1985.

[7] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. *Phrack*, (54), December 1998.

[8] D. Katabi, M. Handley, and C. Rhors. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM SIGCOMM*, August 2002.

[9] Ken Keys, David Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of coralreef: an internet traffic monitoring software suite. PAM2001, Workshop on Passive and Active Measurements, RIPE, 2001.

[10] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM CCR*, 32(3):62–73, July 2002.

[11] David Moore. Personal conversation. also see caida analysis of code-red, 2001. `http://www.caida.org/ analysis/ security/ code-red/`.

[12] Cisco netflow. `http://www.cisco.com /warp /public /732 /Tech /netflow`.

[13] Riverstone Networks. Lfap: Lightweight flow accounting protocol. `http://www.riverstonenet.com/ technology/ accounting_for_profitability.shtml`.

[14] David Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *USENIX LISA*, pages 305–317, December 2000.

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

[16] Stuart Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, (10), 2002.

[17] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM*, September 1998.

[18] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2), 1990.

[19] Ming-Young You and Cheng-Shang Chang. Resampling for wireless access. In *Proceedings of IEEE PIMRC*, June 1996.

APPENDIX

A. AVERAGE ERROR FOR THE VIRTUAL AND MULTIRESOLUTION BITMAPS

We first determine the variance of the estimate for the number of flows hashing to the virtual bitmap $VAR[\widehat{m}]$. The number of flows hashing to the bitmap m is a random variable distributed binomially. For this analysis we will assume that there are two independent hash functions one that decides which flows map to the virtual bitmap and one that decides how those get mapped to individual bits[11]. Therefore the analysis of the direct bitmap from sections 4.1 and 4.2 describes the conditional distribution of the random variable $\widehat{m}$ given the value of m.

$$\begin{aligned}
E[\widehat{m}|m=i] &= i \\
VAR[\widehat{m}|m=i] &\approx b\left(e^{i/b} - i/b - 1\right) \\
E[\widehat{m}^2|m=i] &= E[\widehat{m}|m=i]^2 + VAR[\widehat{m}|m=i] \\
&\approx i^2 + b\left(e^{i/b} - i/b - 1\right)
\end{aligned}$$

To obtain $VAR[\widehat{m}]$ we need $E[\widehat{m}]$ and $E[\widehat{m}^2]$.

$$\begin{aligned}
E[\widehat{m}] &= \sum_{i=0}^{n} P(m=i)E[\widehat{m}|m=i] \\
&= \sum_{i=0}^{n} P(m=i)i = E[m] = \alpha n \\
E[\widehat{m}^2] &= \sum_{i=0}^{n} P(m=i)E[\widehat{m}^2|m=i] \\
&\approx \sum_{i=0}^{n} P(m=i)\left(i^2 + b\left(e^{i/b} - i/b - 1\right)\right) \\
&= E[m^2] + bE\left[e^{m/b}\right] - bE[m/b] - bE[1] \\
&= E[m^2] + bE\left[e^{m/b}\right] - E[m] - b
\end{aligned}$$

To compute $E\left[e^{m/b}\right]$, we use the Taylor expansion of $f(x) = e^{x/b}$ around the value $E[m]$.

[11]Any reasonable hash function will make the two processes independent, so in practice we can use a single hash function.

$$\begin{aligned}
e^{x/b} &= e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}(x - E[m]) \\
&\quad + \frac{1}{2b^2}e^{E[m]/b}(x - E[m])^2 + \dots \\
E\left[e^{m/b}\right] &\approx e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}E[m - E[m]] \\
&= e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}(E[m] - E[m]) = e^{E[m]/b}
\end{aligned}$$

To see how far off we are with this approximation, we compute below the value of the expectation of the third term of the Taylor expansion, the first term we ignore. Its ratio to our approximate result gives some indication of how much we are off. For example with $b = 200$ which is smaller than what we expect to be used in practice and a flow density of $\rho = 8$ which is much above the flow densities virtual bitmaps would be expected to operate accurately in, we are off by less than 2%. We also note here that the contribution of further terms is even smaller because they have higher powers of b at the denominator.

$$\begin{aligned}
E\left[\frac{1}{2b^2}e^{E[m]/b}(x - E[m])^2\right] &= \frac{1}{2b^2}e^{E[m]/b}E\left[(x - E[m])^2\right] \\
&= \frac{1}{2b^2}e^{E[m]/b}VAR[m] \\
&= \frac{1}{2b^2}e^{E[m]/b}n\alpha(1-\alpha) \\
&< \frac{1}{2b^2}e^{E[m]/b}n\alpha = \frac{E[m]}{2b^2}e^{E[m]/b} \\
\frac{E\left[\frac{1}{2b^2}e^{E[m]/b}(x - E[m])^2\right]}{e^{E[m]/b}} &< \frac{E[m]}{2b^2} = \frac{\rho}{2b}
\end{aligned}$$

Now substituting our approximate value for $E[e^{m/b}]$ we get $E[\widehat{m}]$ which we use to compute $VAR[\widehat{m}]$.

$$\begin{aligned}
E[\widehat{m}^2] &\approx E[m^2] + be^{E[m]/b} - E[m] - b \\
VAR[\widehat{m}] &= E[\widehat{m}^2] - E[\widehat{m}]^2 \\
&\approx E[m^2] - E[m]^2 + be^{E[m]/b} - E[m] - b \\
&= VAR[m] + be^{n\alpha/b} - n\alpha - b \\
&= n\alpha(1-\alpha) + be^{\rho} - n\alpha - b \\
&= be^{\rho} - n\alpha^2 - b < b\left(e^{\rho} - 1\right)
\end{aligned}$$

$$\begin{aligned}
SD\left[\frac{\widehat{n}}{n}\right] &= \frac{SD[\widehat{m}]}{n\alpha} \lessapprox \frac{\sqrt{b\left(e^{\rho}-1\right)}}{n\alpha} = \frac{\sqrt{e^{\rho}-1}}{n\alpha/b\sqrt{b}} \\
&= \frac{\sqrt{e^{\rho}-1}}{\rho\sqrt{b}}
\end{aligned}$$

The tightness of the bound depends on the term $n\alpha^2$. Since the whole variance $VAR[\widehat{m}]$ is at least $n\alpha(1-\alpha)$, we are off by a factor of at most $1-\alpha$, therefore if α is small (i.e. the virtual bitmap covers only a small portion of the hash space), this bound is tight, but as α approaches 1 the bound is not tight anymore. Indeed for $\alpha = 1$ we have a direct bitmap whose accuracy is described by Equation 3 which can be significantly lower Equation 4 when the flow density is low.

The multiresolution bitmap bases its estimate of the number of active flows on its estimate $\widehat{m}$ for the number of flows hashing to the base component *and all finer ones.* We use m_b for the number of flows hashing to the base component and m_f for the number of flows hashing to all finer components ($m = m_f + m_b$). For this analysis we do not treat the finer components individually, but replace them with a single component: we extend the next component after the base to cover all the finer components, thus its size increases from b to $bk/(k-1)$. This is equivalent to or-ing together the bits of all the finer components until they are all at the granularity of the first component after the base. The benefit of this simplification is that the result will not depend on the number of finer components, thus it will apply no matter which component we use as base. While it is intuitively obvious that or-ing bits together leads to loss of information and thus increases the variance of $\widehat{m_f}$, we prove it too in the technical report version of this paper [4].

As with the virtual bitmap, we assume that the hash function deciding which component a flow gets mapped to and the two hash functions deciding to which of the bits of the component the flow is mapped are independent. While the sampling errors of $\widehat{m_b}$ and $\widehat{m_f}$ are correlated, the correlation is negative and its value is small when the sampling factor is large, so we ignore it. Because of the independence of mapping flows to bits, the collision errors are uncorrelated.

$$\begin{aligned}
VAR[\widehat{m_b}] &\lessapprox b\left(e^{\rho_b} - 1\right) = b\left(e^{\rho} - 1\right) \\
VAR[\widehat{m_f}] &\lessapprox \frac{bk}{k-1}\left(e^{\rho_f} - 1\right) = \frac{bk}{k-1}\left(e^{\rho/k} - 1\right) \\
VAR[\widehat{m}] &= VAR[\widehat{m_b}] + VAR[\widehat{m_f}] + COV[\widehat{m_b}, \widehat{m_f}] \\
&< VAR[\widehat{m_b}] + VAR[\widehat{m_f}] \\
&\lessapprox b\left(e^{\rho} - 1 + \frac{k}{k-1}\left(e^{\rho/k} - 1\right)\right) \\
&= \frac{bk}{k-1}\left(\frac{k-1}{k}\left(e^{\rho} - 1\right) + e^{\rho/k} - 1\right) \\
SD\left[\frac{\widehat{n}}{n}\right] &= \frac{SD[\widehat{m}]}{n\alpha} = \frac{SD[\widehat{m}]}{\rho bk/(k-1)}
\end{aligned}$$

$$SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{\frac{k-1}{k}\left(e^{\rho} - 1\right) + e^{\rho/k} - 1}}{\rho\sqrt{\frac{bk}{k-1}}} \tag{6}$$

Is the error introduced by collapsing all finer components into a single one acceptable? To answer this question we derived two formulas similar to Equation 6: one that maintains one finer component and collapses all the rest (thus working with 3 bitmaps) and one maintaining two finer bitmaps and collapsing the rest (thus working with 4 bitmaps). We plugged in all three formulas into the algorithm for computing the sizes of the components of the multiresolution bitmaps. The more accurate (and more complicated) formulas always resulted in lower sizes for the bitmaps, but the differences were significant only for low values of k. Thus the formula using 3 bitmaps reduces the bitmap size with respect to Equation 6 by 3% for $k = 2$ and 1% for $k = 3$. Using 4 bitmaps reduces the bitmap size (with respect to the formula with 3 bitmaps) by less than 1% even for $k = 2$. As a tradeoff between accuracy and simplicity we decided to use in this paper the formula derived based on 3 bitmaps which is Equation 5 from Section 4.2.