

UC San Diego

Technical Reports

Title

Counting the number of active flows on a high speed link

Permalink

<https://escholarship.org/uc/item/9gs7d82n>

Authors

Estan, Cristian
Varghese, George
Fisk, Mike

Publication Date

2002-05-21

Peer reviewed

Counting the number of active flows on a high speed link

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

May 21, 2002

Abstract

In this paper we 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. The central difficulty is that count processing must be done in a packet arrival time (8 nsec at OC-768 speeds) and hence must take a small number of memory references to limited, fast memory. A naive solution that maintains a hash table requires several Mbytes because the number of flows can be more than a million. By contrast, our new 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 used one of our new algorithms to replace the port scan detection component of the popular intrusion detection system Snort. Doing so reduced the memory usage on a ten minute trace from 51 Mbytes to 5.7 Mbytes while maintaining a 99.78% probability of alarming on a scan within 6 seconds of when the large-memory algorithm would alarm. By contrast, the best known prior algorithm (probabilistic counting) takes 4 times more memory on the port scan application and 8 times more memory 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.

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 up to OC-768 speeds (40 Gigabits/second) are current-

ly being developed. While the main bottlenecks (e.g., lookups, classification, QoS) 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.

Towards this goal, this paper isolates and provides solutions for an important problem that occurs in various networking applications: *counting the number of active flows in packets received on a link in a specified time period*. 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 in a specified time period. Our algorithms measure the number of active flows using very small memory that can easily be stored in on-chip SRAM or even processor registers. By contrast, the naive algorithms we describe below would require massive amounts of (slow) DRAM.

For example, the 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 a million flows [3] today, this naive scheme would minimally require 8 Mbytes of high speed memory.¹ Such large SRAM memory is expensive or not feasible for a modern router.

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

By contrast, in this paper we will describe a randomized scheme called *adaptive bitmap*, that 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. We will also describe a scheme called *triggered bitmap* (that is optimized for running multiple concurrent applications of the counting problem many of which have small counts) which 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 fields². 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. An alternate way of defining the problem without using measurement interval is by counting the flows that have at least one packet in a queue that packets are added to and removed from dynamically. In this paper we will 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. Thus we seek solutions that use a very small amount of memory, use only 1 or 2 memory accesses,³ and have high accuracy. In particular, we examine the tradeoff between memory usage and average error.

²We 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).

³Actually, larger numbers of memory accesses are perfectly feasible at high speeds using SRAM and pipelining; however, in order to compare the tradeoff between memory and accuracy we prefer to keep the number of memory accesses to be no more than that of the naive scheme.

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. The widely deployed Snort IDS[9] uses the naive approach of storing a record for each active connection. This is 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 and not the actual IPs and ports the connections were made to, we don't need to keep a record for each connection either. 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 is able to 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. Such scalability is particularly important in the context of the recent race to provide wire speed intrusion detection [1].

Estimating the Spreading Rate of a Worm: During Aug 1 - Aug, while trying to track the recent Code Red worm [7]. 12, collecting packet headers for Code Red traffic on a /8 network produced 0.5GB 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).

Detecting denial of service attacks: In [6] Mahajan et al. propose a mechanism that allows backbone routers to limit the effect of (distributed) DoS attacks. While the mechanism assumes that these routers can detect an ongoing attack it does not give a concrete algorithm for it. In [2] Eitan and Varghese discuss a number of algorithms that can detect destination addresses or prefixes that receive large amounts of traffic. While this 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. If for each of the suspected victims 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 denial of service attack is in progress.

General measurement: Often counting the number of distinct values in given header fields can provide useful data. For example one could measure the *number* of sources using a protocol version or variant to get an accurate image of protocol deployment. Another example is dimensioning the various caches in routers which benefits from prior measurements of typical workload. Such caches include: packet classification caches, multicast route caches for Source-Group (S-G) state, and ARP caches.

Packet scheduling: Many scheduling algorithms try to ensure that all flows can send at the current “fair share” of the available bandwidth. While there are scheduling algorithms that compute the fair share without using per-flow state (e.g. CSFQ [11]), they require explicit cooperation between edge and core routers. 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.

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 [4], introduced in the context of databases. We will use probabilistic counting as a base to compare our algorithms against. Whang et al address the same problem in [12] and propose an algorithm that is equivalent to the simplest (direct bitmap) algorithm we describe.

The insight behind probabilistic counting is to compute a metric of how uncommon a certain record is and keep track of the “uncommonness” across all records. If we see very uncommon records, the algorithm concludes it saw a large number of records. More precisely, for each record the algorithm computes a hash function on L bits. It then counts the number of consecutive zeroes starting from the least significant position of the hash result. It then stores in a bitmap the numbers seen so far. For example if the algorithm has seen records with 1,2,3 and 5 consecutive zeroes, the accumulated bitmap would

be 111010000. For this bitmap, the algorithm computes the metric of uncommonness $r = 3$ (not 5 which is an outlier). The final estimate for the number of flows is $n = \alpha 2^r$, where α is a “magic constant” that represents a statistical correction factor. This basic form can guarantee at most 50% accuracy. By dividing the hash values into *nmap* groups and averaging over the estimates for the count provided by each of them, probabilistic counting reduces the error of its final estimate. We will describe a family of algorithms that outperform 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 [8] or LFAP that report per flow records for very fine grained flows. This is useful for traffic engineering. The information can be used by to count flows, but is not optimized for such a purpose. Besides the large amount of memory needed, in modern high speed routers *updating* state on every packet arrival is infeasible at high speeds. Ideally, such 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 speed DRAM which slows down router processing. For example, Cisco recommends the use of sampling at speeds above OC-3: only the sampled packets result in updates to the flow cache that keeps the per flow state. Unfortunately, sampling has problems of its own since it affects the accuracy of the measurement data. Sampling works reasonably for estimating the traffic sent by large flows, but has extremely poor accuracy for estimating the number of flows. This is because uniform sampling will tend to produce more samples of flows that send more traffic, thereby biasing any simple estimator that counts the number of flows in the sample and multiplies this count by the inverse of the sampling rate.

The Snort [9] intrusion detection system (IDS) uses a similar memory-intensive approach to detect port scans: it maintains a record for each active connection and a connection counter for each source IP. More elaborate algorithms have been used 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 described in [13] uses techniques equivalent to our direct bitmap and virtual bitmap to estimate this number, but has no equivalent of our more 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 through experiments on traces that our estimates are close to actual values. The family contains three core algorithms and five derived algorithms. Even though the first two core algorithms (direct and virtual bitmap) have been invented earlier, 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 five 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 with a correction factor that takes into account hash “collisions”⁴. Even with this

⁴We assume in our analysis that the hash function distributes the

correction factor, the algorithm becomes very inaccurate when the number of flows is much larger than the number of bits in the bitmap. The only way to preserve accuracy is to have a bitmap size that scales 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 1) and extrapolates 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 the number of flows differs significantly from what we configured it for, the estimates are inaccurate. While in general one wants an algorithm that is accurate over a wider range, we note that even an unadorned virtual bit map is useful. For example, consider a security application where we wish to set an alarm when the number of flows crosses a threshold. The virtual bitmap can be tuned for this threshold.

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 we can achieve an average error of 3% with 292 bytes, irrespective of the number of flows, as long as the sampling factor is set to an optimal value.

3.3 Multiresolution bitmap

Virtual bitmap is simple to implement, uses little memory and gives very accurate results. But it does require 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 different anticipated ranges of the number of active flows (different

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.

“resolutions”). The union of all these ranges is chosen to cover all possible values for the number of flows. 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. *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 1 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 virtual bitmap operates. We can consider that instead of generating an integer between 0 and the size of the virtual bitmap, the hash function covers a continuous interval. The virtual bitmap covers the full interval while the physical bitmap covers a smaller portion (the ratio of the sizes of the two is the sampling factor of the virtual bitmap). We divide the interval corresponding to the physical bitmap into equal sized sub-intervals, each corresponding to a bit. A bit in the physical 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 1.

A multiresolution bitmap is essentially a combination of multiple bitmaps of different “resolutions”, such that for each packet only the highest resolution bitmap it maps to is updated. Thus each bitmap loses a small 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).

In Section 4.3 we answer questions such as what is the best ratio between the resolutions of neighboring components, how many bits we need in each etc. In Appendix E we compare our multiresolution bitmap to probabilistic counting showing that while that 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. 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. Assuming “quasi-stationarity”, the algorithm will be 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 will provide 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 of the expected resolution are replaced with a single large component consisting of a virtual bitmap. The location of the virtual bitmap within the multiresolution bitmap 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. The update of the bitmap happens exactly as in the case of the multiresolution bitmap, except that

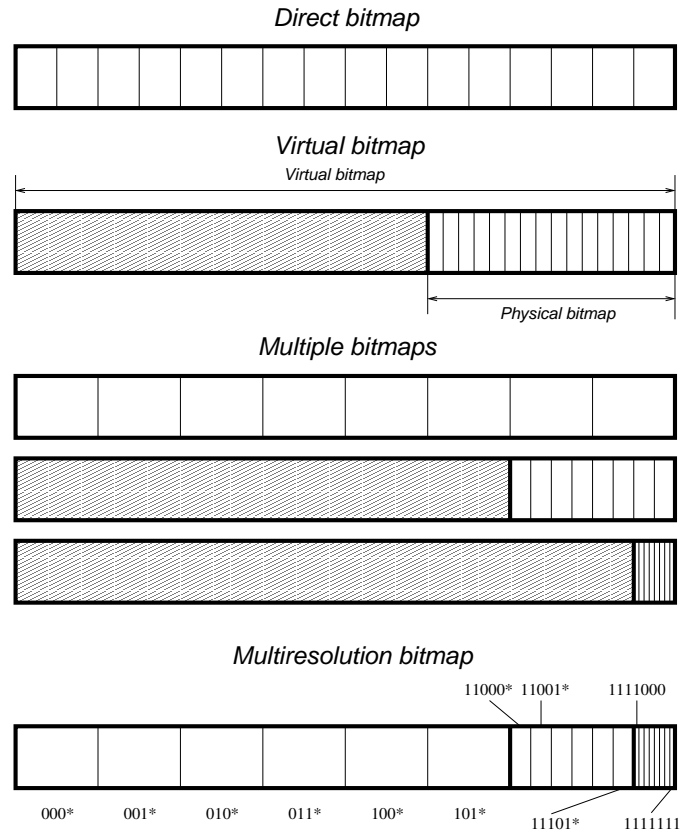


Figure 1: The multiresolution bitmap from this example uses a single 7-bit hash function to decide which bit to map a flow to

the logic is changed slightly when the hash value maps to the virtual bitmap component.

for that source and it's 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.

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 all of sources 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

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 then 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 will not 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 said 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. We only note that all our algorithms can be modified to handle this case by replacing every bits with a counter. 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 used the number of zero bits in the case with measurement intervals. A counter will be zero if and only if no active flows map to it.

4 Algorithm Analysis

In this section we provide approximate analyses for 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 average error. 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 \hat{n} for the number of active flows.

$$\hat{n} = b \ln \left(\frac{b}{z} \right) \quad (1)$$

Virtual bitmap: Let v be the size of the virtual bitmap and b the number of bits of physical memory used. We use $\alpha = b/v$ for the sampling factor (the fraction of the virtual bitmap that is covered by the physical bitmap). The probability for a given flow to hash to the physical bitmap is equal to the sampling factor $p_{ph} = \alpha$. Let m be the number of flows that actually hash to the physical bitmap. Its probability distribution is binomial with an expected value of $E[m] = \alpha n$. By substituting in Equation 1, we obtain Equation 2 for the estimate of the number of active flows.

$$\hat{n} = \frac{1}{\alpha} b \ln \left(\frac{b}{z} \right) = v \ln \left(\frac{b}{z} \right) \quad (2)$$

Multiresolution bitmap: The multiresolution bitmap is a combination of many components, each tuned to provide accurate estimates over a particular range. But 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 the coarsest component that has no more than set_{max} bits set as the base component in lines 1 to 5 of Figure 2. set_{max} is a precomputed threshold set so that we choose the component that gives us the most accurate result. 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 2). To obtain the result we only need to apply the correction corresponding to the sampling factor (lines 18 and 19). Other parameters used by this algorithm are the 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 only difference is that we use different thresholds for the big component. 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


```

ESTIMATEFLOWCOUNT
1  base = c - 1
2  while bitsSet(component[base]) < setmax and base > 0
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
7          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 + blast ln(blast/bitsZero(component[c]))
18 factor = kbase-1
19 return factor * m

```

Figure 2: Algorithm for computing the estimate of the number of active flows for a multiresolution bitmap

total number of bits in the direct bitmap. If the multiresolution bitmap is deployed, we use the algorithm from Figure 2 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 deviation of the estimate \hat{n} of the number of active flows n . Our measure of accuracy is the average (relative) error \hat{n}/n . One parameter that is useful in these analyses is the flow density ρ which gives the average number of flows that hash to a bit.

Direct bitmap: Since the bits in the bitmaps are set almost independently⁵ with the same probability, we can model the bits as independent Bernoulli trials. Thus we get Equation 3 for the average error of a direct bitmap. The error in our estimates occurs because we can't always guess right about the number of collisions that occurred. Equation 3 is derived for large values of n and b of the same order of magnitude. When n is small this bound is not tight.

⁵Appendix B has a more detailed discussion about why and when it is valid to rely on this assumption

$$\frac{SD[\hat{n}]}{n} \approx \frac{\sqrt{b}}{n} \sqrt{e^{n/b} - 1} = \frac{\sqrt{b}}{n} \sqrt{e^\rho - 1} \quad (3)$$

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 B we analyze these two errors. 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.

$$\frac{SD[\hat{n}]}{n} \approx \frac{\sqrt{e^\rho + \rho - 1 - \frac{b}{n}\rho^2}}{\rho\sqrt{b}} < \frac{\sqrt{e^\rho + \rho - 1}}{\rho\sqrt{b}} \quad (4)$$

Multiresolution bitmap: To compute the average error of the estimate of the multiresolution bitmap, we could take into account the collision errors of all finer components. This would result for a different formula for each component. Equation 5 is a slightly weaker bound that holds for all components but the last one as long as

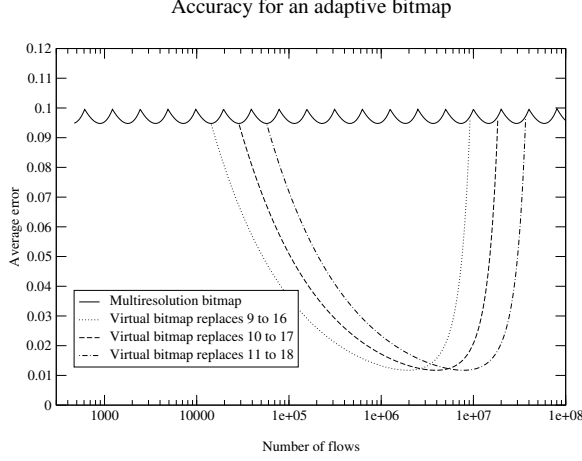


Figure 3: 8 of the 19 components of the multiresolution bitmap are replaced by the virtual bitmap of size 15208 ($b=94$)

the number of bits in the last component b_{last} is at least $b/(1 - 1/k)$. Based on this formula we compute numerically the flow density ρ_{max} where estimates based on a given component are still more accurate than those based on the next one (whose flow density is $\rho' = \rho/k$) and thus obtain $set_{max} = b(1 - e^{-\rho_{max}})$. For the last component of the multiresolution bitmap we can use Equation 4 directly.

$$\frac{SD[\hat{n}]}{n} \approx \frac{\sqrt{(1 - \frac{1}{k})(e^\rho - 1) + e^{\rho/k} + \rho - 1}}{\rho \sqrt{b/(1 - \frac{1}{k})}} \quad (5)$$

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 3 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	$> \frac{N}{1.5 + \sqrt{6A^2N - 3.75}}$
Virtual bitmap	$\frac{2.097}{A^2}$
Multiresolution bmp.	$\frac{0.87}{A^2} (\log_2(A^2N))$
Adaptive bitmap	$\approx \frac{2.097}{A^2}$

Table 1: The size of the direct bitmap scales worse than \sqrt{N}/A , the size for the virtual bitmap depends only on the square of the average error, the size of the multiresolution bitmap depends on the square of the average error times the logarithm and under certain assumptions, the adaptive bitmap delivers the accuracy of the virtual bitmap while dynamically adapting 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. 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 error A . We base our computations on the formulas of the previous section.

Direct bitmap: We can see that if n increases very much the exponential takes over and dominates the error in Equation 3. So the size of the bitmap scales almost linearly with N . To obtain the proper lower bound from Table 1, we approximate $A^2N \approx (e^\rho - 1)/\rho > (1 + \rho + \rho^2/2! + \rho^3/3! - 1)/\rho = 1 + \rho/2 + \rho^2/6$.

Virtual bitmap: The error of the virtual bitmap (Equation 4) is minimized by a certain value of the flow density. Using simple calculus we find that $\rho_{optimal} = 2$ and this corresponds to 13.5% of the bits of the bitmap being not set. By substituting, we obtain the error for this “sweet spot” in Equation 6. 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 exactly 2. For this application, if we have 210 bits, the standard deviation of our estimate is at most 10% of the actual number of flows, no matter how large N is. If we have 2,331, the standard deviation is at most 3% of the actual number of flows, and if we have 29,075 it is at most 1%. If we want the virtual bitmap to have at most a certain error for 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 $\rho_{max}/\rho_{min} = N_{max}/N_{min}$

r	$bigb/b$	improvement
2	3.1824	1.2069
3	5.3405	1.5187
4	9.1712	1.9603
5	16.0633	2.5745
6	28.5916	3.4216
7	51.5673	4.5862
8	93.9867	6.1856
9	172.7982	8.3832
10	319.9593	11.4046
11	596.0726	15.5644
12	1116.0278	21.2958

Table 2: The relation between the number of components covered and the ratio of the bitmap sizes and the improvement in steady state average error as compared to the worst case

and ρ_{min} and ρ_{max} produce the same error. Once we have these values, we can compute the sampling factor and the number of bits.

$$\frac{SD[\hat{r}]}{n} \approx \frac{1.44826}{\sqrt{b}} \quad (6)$$

Multiresolution bitmap: For the multiresolution bitmap, we have to ensure that the average error of the component doesn't exceed the desired value as the flow density increases from a certain ρ_{max} to a ρ_{max} , where we know that $\rho_{max}/\rho_{min} = k$ (k is the ratio between the resolutions of neighboring components). Numerical computations from Appendix D show that if the number of components is large enough $k = 2$ is always the choice that results in the smallest memory consumption. We also obtain the number of bits per component $b = 0.8687/A^2$, the number of components needed $c = 2 + \lceil \log_2(N/(5.6742 b)) \rceil$, $\rho_{max} = 3.2426$ and the parameter we use to choose the base component $set_{max} = 0.9609 * b$. By assuming the last component has $b_{last} = 2 * b$ bits, we obtain the total memory usage reported in Table 1.

Adaptive bitmap: For brevity we omit the detailed discussion of the configuration of the adaptive bitmap. In Table 2 we report the costs and benefits of the adaptive bitmap. The first column lists the number 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) in order to insure that the adaptive bitmap never has a worse average error than the original multiresolu-

Name	No. of flows (min/avg/max)	Length (s)	Encr.
MAG+	93,437/98,424/105,814	4515	no
MAG	97,409/99,225/100,339	90	no
IND	13,746/14,349/14,936	90	yes
COS	5,157/5,497/5,784	90	yes

Table 3: The traces used for our measurements

tion 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 will be 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 towards 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 from an OC-48 backbone link and two encrypted traces from the MOAT project of NLANR from 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 someone would use when 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 source-destination IP address and port 5-tuples. Table 3 gives a summary description of the traces we used. All algorithms used equivalent CRC-based hash functions with random generator functions.

5.1 Virtual bitmap

We postpone till Appendix C the measurements that confirm the validity of Equation 3 and the conclusion that the best average error for a virtual bitmap algorithm is achieved when the flow density is $\rho = 2$. Actually Equation 3 proves to be conservative in the sense that actual errors are somewhat smaller than predicted by the formula.

Our third set of measurements confirms our formula for the accuracy of the virtual bitmap 6 and compares it to probabilistic counting using the same amount of memory. We have three configurations, with different amounts of memory, based on the error predicted by Equation 6: with 210 bits (10% error, results in table 4), 2,331 bits (3% error, results in table 5) and 20,975 bits (1% error, results in table 6).

For each trace, we configure the virtual bitmap so that the flow density is (on average) $\rho = 2$. For the third configuration, we don't use the virtual bitmap for the traces IND and COS because they have much fewer flows than the number of bits available, so we use direct bitmap. Probabilistic counting is configured to handle up to 100,000 flows. We use the following three configurations for probabilistic counting: $nmap = 12$ bitmaps of $L = 18$ bits each, 167 bitmaps of 14 bits and 2,098 bitmaps of 10 bits each.

For each configuration we use 20 runs of both algorithms, with different hash functions. In the tables we report the accuracy (computed as the square root of the average of the squares of the relative errors), and the largest errors in both directions (most severe underestimation and the most severe overestimation) over the 18×20 measurement intervals considered. Note that in all experiments the virtual bitmap is more accurate than probabilistic counting. We explain the surprisingly bad results of probabilistic counting on the COS trace using 20,975 bits by referring to the inaccuracy of this algorithm for small values.

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 decided to use a synthetic trace for this ex-

periment. 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 performed 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 performed 400 runs for each configuration of both algorithms with different hash functions.

Figures 7 to 9 show the results of the experiments. We can see that in all three experiments, the average error of the multiresolution bitmap is better than predicted, especially for small values. We explain the periodic "fluctuations" of average error from table 7 by the variability of average error of the components. The peaks correspond to where components are least accurate and hand off to each other. The peaks are more pronounced in this table than the others because due to the small number of bits in each component, it happens more often that not the best component is used as a base for the estimation.

Probabilistic counting is worse than the multiresolution bitmap, especially for small values. We show in the full version of the paper that the data collected by the two algorithms is equivalent, so it might be surprising that their accuracies are so different. We attribute the big errors of probabilistic counting for low values to the way it evaluates the collected data and the worse error for higher values for the suboptimal dimensioning of the algorithm (as recommended in [4]).

5.3 Adaptive bitmap

The experiments from this section compare adaptive bitmap to virtual bitmap, multiresolution bitmap and probabilistic counting on a three traces. The results are presented in table 10. 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 $k = 2$, $b = 188$, $c = 19$. The virtual bitmap component is 15,208 bytes large and replaces 8 components of the

Trace	Virtual bitmap (min/avg/max)	Probabilistic counting (min/avg/max)
MAG	-18.973/8.942/33.067%	-47.365/21.4348/65.587%
IND	-19.063/9.178/26.481%	-47.696/24.807/100.391%
COS	-20.011/8.909/28.041%	-40.990/24.262/87.364%

Figure 4: Results with 210 bits (expected error 10%)

Trace	Virtual bitmap (min/avg/max)	Probabilistic counting (min/avg/max)
MAG	-5.891/2.619/8.115%	-15.513/5.894/19.055%
IND	-6.321/2.468/7.514%	-14.471/5.800/19.139%
COS	-6.562/2.257/6.492%	-16.182/5.363/15.727%

Figure 5: Results with 2331 bits (expected error 3%)

Trace	Virtual bitmap (min/avg/max)	Probabilistic counting (min/avg/max)
MAG	-2.320/0.782/2.451%	-4.872/1.636/4.569%
IND	-1.652/0.575/1.505%	-3.444/1.367/3.732%
COS	-1.583/0.501/1.267%	7.666/12.919/16.973%

Figure 6: Results with 20975 bits (expected error 1%)

Trace	Virtual bitmap	Adaptive bitmap	Multi-Resolution bitmap	Probabilistic counting
MAG+	-3.285/0.932/3.545%	-3.297/0.943/3.583%	-9.374/2.272/8.233%	-10.533/2.840/11.494%
IND	-1.692/0.639/1.778%	-2.427/0.696/2.063%	-7.597/2.070/5.023%	-7.943/2.861/8.810%
COS	-1.454/0.579/1.720%	-1.735/0.623/1.726%	-6.077/2.468/6.571%	-6.050/2.464/6.463%

Figure 10: Comparison of algorithms using 16Kbits of memory

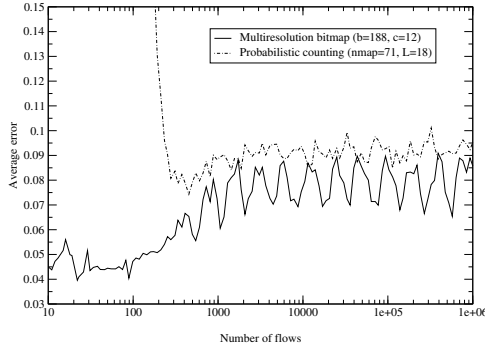


Figure 7: Configured for an average error of 10%

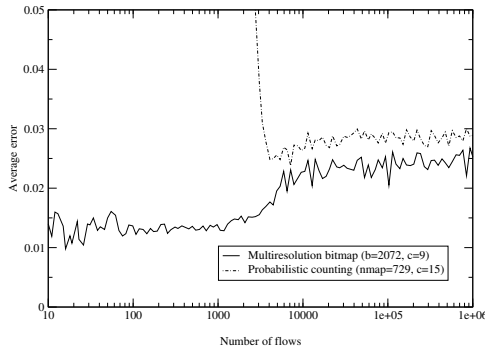


Figure 8: Configured for an average error of 3%

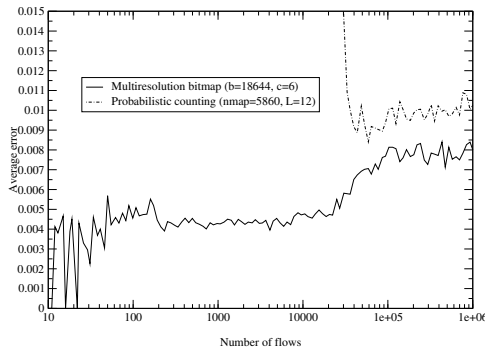


Figure 9: Configured for an average error of 1%

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. We can see that adaptive bitmap is roughly 2.5 times more accurate than probabilistic counting. We were only able to achieve the same error as this adaptive bitmap with a probabilistic counting algorithm that used much more (128 Kbits) of memory. The major message here is that an adaptive bitmap can achieve almost the same benefits of virtual bitmap (e.g., ordinary of magnitude reduction in memory for same accuracy) when the number of flows does not vary dramatically, as seems common in networking applications.

We ran virtual bitmap in two configurations: the first tuned to have the smallest possible error for the average number of active flows and the second to have the smallest possible error for the smallest and largest number of active flows in the trace. Since the results were very similar, we report only the first configuration in column 1. For the smaller traces we used direct bitmap because in 16 Kbits, there were no benefits for virtual bitmap. The multi-resolution bitmap configuration was $k = 2$, $b = 990$ and $c = 15$. The expected accuracy is 3.069%.

We can see that the performance of the adaptive bitmap is only slightly worse than that of the virtual bitmap, even in the case of the two small traces where the virtual bitmap is replaced by direct bitmap. The gain in robustness more than compensates for this small loss of accuracy. While the multi-resolution bitmap performs better than predicted by our analysis and better than probabilistic counting, it is still roughly 2.5 times less accurate than the adaptive bitmap.

5.4 Triggered bitmap

So far, all our measurements have focused on one instance of the counting problem. We have shown that virtual bitmap and adaptive bit map can provide order of magnitude improvements in the memory required for counting compared to probabilistic counting, assuming that the count range is known in advance or if the count value is quasi-stationary. The reader may, however, have two objections. First, our results so far do not indicate how an actual application could benefit. Second, while a memory reduction from 128 Kbits to 16 kbits is impressive in a relative sense, we have not demonstrated that it is important in an absolute sense because 128 Kbits of S-RAM is easily available on-chip or in a processor cache.

Measurement interval	Snort	Prob. count.	Triggered bmp.
12 sec	1,846K	2,364K	472K
600 sec	52,107K	23,251K	5,828K

Table 4: The memory usage of various port scan detection algorithms (in Kbytes)

To remedy these two problems we describe experiments that indicate benefits on a real port-scanning application. At the same time, the reduction in memory is important in an absolute sense because we need to maintain a large number (one per source) instances of the counting problem.

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 last 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 16.5% for up to 283,352 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 performed 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 was⁶ 13.58%.

Also, our algorithm reported 84.28% of the sources with 4 connections as reaching the threshold, 97.85% of those with 5, and all (100%) of the sources that had at least 9 connections. The memory usages for the two configurations are reported in Table 4. In the table we report the *maximum* of triggered bitmap over the 20 runs. We can see that the triggered bitmap uses roughly 4 times less memory 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 that 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 possible undercounting. However, the probability of undercounting decreases exponentially with the number of events. For example, the probability is 97.85% at 5, 99.78% at 6, 99.97% 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 will detect a continuing scan with probability 97.85% within 3 seconds and 99.78% within 6 seconds. Note also that port scans are usually the result of a brute-force network exploration such as Nmap [5], or Code Red [7]. 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.

Finally, note that because our algorithms use as much as an order of magnitude less memory per source, they also enable stealthy slow scans to be detected 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. By doing so, we can detect more stealthy port scans, a goal of many detection systems [10].

⁶This is an average over all sources. We did notice some “peculiarities”: for sources that had 4,5 and 8 connections the average errors were around 11%, for sources with 6 connections it was 18% while for all others the averages were around 15%. We explain these as effects of having such a small direct bitmap.

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

6 Conclusions

In this paper we show that, 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. 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 other algorithms are introduced for the first time in this paper.

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 allow them 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 order of magnitude lower error when the number of flows is small and is only slightly worse 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 5) that application 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 292 bytes to achieve an error of 2.619% compared to 5,200 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 5.7 Mbytes on a 10 minute trace compared to the 51 Mbytes required by the naive algorithm and 23 Mbytes required by probabilistic counting. 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, K. Claffy, Stefan Savage and Florin Baboescu for extremely valuable conversations. We would also like to thank the anonymous reviewer whose comments helped make this paper better. This work was made possible by a grant from NIST for the Sensilla Project.

References

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

- [2] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM*, August 2002.
- [3] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM*, December 1999.
- [4] 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.
- [5] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. *Phrack*, (54), December 1998.
- [6] Ratul Mahajan, Steve M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. <http://www.aciri.org/pushback/>, July 2001.
- [7] David Moore. Personal conversation. also see caida analysis of code-red, 2001. <http://www.caida.org/analysis/security/code-red/>.
- [8] Cisco netflow. <http://www.cisco.com/warp/public/732/Tech/netflow>.
- [9] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.
- [10] Stuart Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. *To appear in Journal of Computer Security*, 2001.
- [11] Ion Stoica, Scott Shenker, and Hui 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.
- [12] 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):208–229, 1990.
- [13] Ming-Young You and Cheng-Shang Chang. Resampling for wireless access. In *Proceedings of IEEE PIMRC*, June 1996.

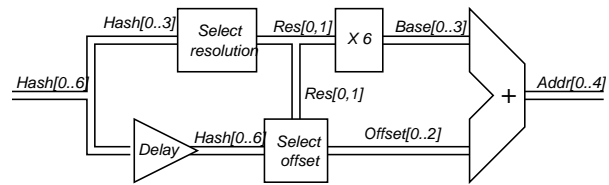


Figure 11: Hardware for selecting the bit to be set

A A hardware implementation for the multiresolution bitmap

While the analysis of the statistical behavior of multiresolution bitmap can look complicated, its implementation is simple. It simplifies the hardware implementation if the size of the components and the ratio of the resolutions are powers of two. Thus for the bottom multiresolution example in Figure 1 one can map the incoming packets to the proper sub-interval by computing a 7-bit hash function and using simple additional combinatorial logic. If the first two bits of the hash are not “11”, the first 3 bits decide which of the sub-intervals in the coarsest component the packet maps to. Otherwise if the third and fourth bit are not 11 (but the first two are), bits from 3 to 5 decide which sub-interval in the intermediate component the packet maps to. If the first 4 bits are 1, bits from 5 to 7 map the packet to the appropriate subinterval in the finest component.

Figure 11 presents a possible hardware implementation for the logic circuit that computes the address of the bit the current packet maps to. It is based on the operations described in the previous paragraph. The input to the circuit is the 7 bit hash function Hash[0..6] that is generated based on the flow ID of the packet. Its output is a 5 bit address Addr[0..4] of the bit that will be set to 1. The leftmost bit in the multiresolution bitmap of figure 1 has an address of 0 and the rightmost has an address of 19.

The “select resolution” block selects the component with the appropriate resolution based on the first 4 bits of the hash. If the coarsest component is used, the output Res[0,1] will have a value of 0 and if the finest component is used its value will be 2. This block can be implemented with few gates. The “X 6” block multiplies this value by 6 because there are 6 bits in each component. Since this block performs multiplication with a constant value, it can be implemented using an adder and a shift register. The “Select offset” block selects which of the bits within the hash to be used to find the offset Offset[0..2] of the bit within the component. In our case it

can be implemented with a 16:4 multiplexer. The final adder adds together the base and the offset to obtain the address of the bit.

B Analysis Details

We first show the derivation of Equation 3. Equation 1 shows us that $\hat{n} = f(z)$ where $f(x) = b \ln(b/x)$. We approximate $SD[\hat{n}] \approx |f'(E[z])SD[z]|$. For the purpose of this analysis we assume that the bits are set independently⁷ with probability $1 - (1-1/b)^n \approx 1 - 1/e^{n/b}$. This gives us a binomial distribution for z . Based on our model, $E[z] \approx b(1/e)^{n/b}$ and $SD[z] \approx \sqrt{b(1/e)^{n/b}(1 - (1/e)^{n/b})}$. Through calculus we obtain $f'(x) = -b/x$. By substituting we get $SD[\hat{n}] \approx b/(b(1/e)^{n/b}) \sqrt{b(1/e)^{n/b}(1 - (1/e)^{n/b})} = e^{n/b} \sqrt{b(1/e)^{n/b}(1 - (1/e)^{n/b})} = \sqrt{be^{n/b}(1 - (1/e)^{n/b})} = \sqrt{b(e^{n/b} - 1)}$. From here Equation 3 is immediate.

For Equation 4 we start from $SD[error_m] = \frac{\sqrt{n(1-\alpha)/\alpha}}{(n/\sqrt{b\rho})\sqrt{(1-b\rho/n)}} = \frac{\sqrt{n(1-b\rho/n)n/(b\rho)}}{n/(\rho\sqrt{b})\sqrt{\rho(1-b\rho/n)}}$ and $SD[error_z] \approx \frac{n/(\rho\sqrt{b})\sqrt{e^\rho - 1}}{n/(\rho\sqrt{b})\sqrt{e^\rho - 1 + \rho(1 - n\rho/b)}}$. We obtain $SD[\hat{n}] = \frac{\sqrt{SD[error_z]^2 + SD[error_m]^2}}{n/(\rho\sqrt{b})\sqrt{e^\rho - 1 + \rho(1 - n\rho/b)}} \approx \frac{\sqrt{SD[error_z]^2 + SD[error_m]^2}}{n/(\rho\sqrt{b})\sqrt{e^\rho + \rho - 1 - (b/n)\rho^2}}$. From here Equation 4 is immediate.

For Equation 5, we focus on $SD[\hat{m}] = \alpha SD[error_z]$. Let m_1 be the number of flows that hash to the coarse component and m_2 be the number of flows that hash to the fine component. Since we use $m = m_1 + m_2$, we have $error_z = error_{z_1} + error_{z_2}$ and since the collision errors for m_1 and m_2 are independent random variables with an expectancy of 0, we have $SD[\hat{m}] = \sqrt{SD[\hat{m}_1] + SD[\hat{m}_2]}$. By the multi-resolution bitmap algorithm we know that $E[m_1] = (1 - 1/k)m$ and $E[m_2] = (1/k)m$.

We will approximate the standard deviations of the collision errors $error_{z_1}$ and $error_{z_2}$ by their values in the case where we knew exactly that $m_1 = E[m_1]$ and $m_2 = E[m_2]$. By applying Equation 3 directly we ob-

⁷The bits are actually not set independently, even assuming a perfect hash function. If we define X_i to be the indicator variable for bit i being set, the correlation two such variables is $corr(X_i, X_j) = -\frac{(1-2/b+1/b^2)^n - (1-2/b)^n}{(1-1/b)^n(1-(1-1/b)^n)} > -1/(b-1)$. Since the correlation is close to 0 when b is large, we conjecture that our model does not introduce significant errors.

tain $SD[\hat{m}_1] \approx \sqrt{(1-1/k)b(e^{m_1/((1-1/k)b)} - 1)}$ and $SD[\hat{m}_2] \approx \sqrt{b(e^{m_2/b} - 1)}$. By using $\rho = m/b$ and the formulae for the expected values of m_1 and m_2 we can eliminate m_1 and m_2 from the formulae. We obtain $SD[\hat{m}_1] \approx \sqrt{(1-1/k)b(e^\rho - 1)}$ and $SD[\hat{m}_2] \approx \sqrt{b(e^{\rho/k} - 1)}$.

By substituting in the formula for the standard deviation of our estimate of m we obtain $SD[\hat{m}] \approx \sqrt{(1-1/k)b(e^\rho - 1) + b(e^{\rho/k} - 1)} = \sqrt{b((1-1/k)(e^\rho - 1) + e^{\rho/k} - 1)}$. From this we obtain $SD[error_z] = 1/\alpha SD[\hat{m}] = n\sqrt{((1-1/k)(e^\rho - 1) + e^{\rho/k} - 1)}/\rho\sqrt{b}$. By combining this with $SD[error_m] = n/(\rho\sqrt{b})\sqrt{\rho(1-b\rho/n)}$ we obtain $SD[\hat{n}] = \frac{\sqrt{SD[error_z]^2 + SD[error_m]^2}}{n\sqrt{((1-1/k)(e^\rho - 1) + e^{\rho/k} - 1) + \rho(1-b\rho/n)}/\rho\sqrt{b}} < n\sqrt{((1-1/k)(e^\rho - 1) + e^{\rho/k} - 1 + \rho)}/\rho\sqrt{b}$. From here Equation 5 is immediate.

While the independence assumption and the use of a Taylor series approximation using derivatives is somewhat cavalier to say the least, the approximations work very well in practice, as indicated by measurements we show next.

C Verifying the analysis

The purpose of our first set of measurements is to verify how well Equation 3 predicts the accuracy of the direct bitmap algorithm. For each measurement interval, we compute the relative error of the estimator, and we compute the square root of the average of the squares of these over all the measurement intervals; we repeat the experiment 10 times with 10 different hash functions. We performed measurements for bitmaps sizes ranging from 0.25 times the number of flows to 10 times the number of flows (the x axes are log scale).

Figures 12, 13 and 14 show our results for the 3 traces we considered. In all these figures we also plotted the values predicted by Equation 3. For all traces and all sizes of the bitmaps, the formula predicts a larger error than what the actual experiments report. We can see that as the size of the bitmap increases, the estimates are much more accurate than predicted because the formula asymptotically approaches $1/\sqrt{n}$ while the actual estimate gets more accurate as the probability of two flows hashing to the same bit decreases.

The purpose of our second set of measurements is to

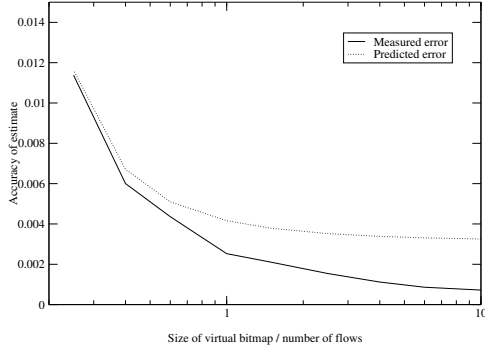


Figure 12: Effect of bitmap size (MAG)

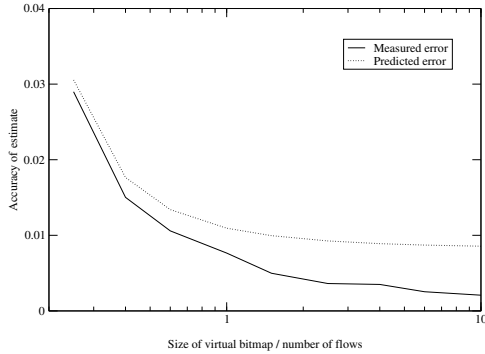


Figure 13: Effect of bitmap size (IND)

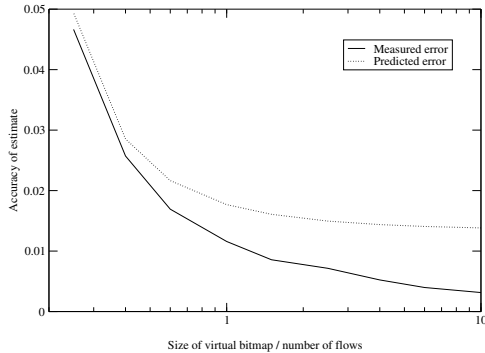


Figure 14: Effect of bitmap size (COS)

verify what the best flow density is for our virtual bitmap algorithm. For all three of the traces we used we performed experiments with physical bitmap sizes 4, 20 and 100 times smaller than the number of flows (averaged out over all measurement intervals). For each of these bitmap sizes, we varied the flow densities⁸ from 0.4 to 4 (we stopped at lower densities if there were any intervals where all bits were set). For each configuration we perform 100 runs with different hash functions. Figures 15 through 23 show our results. We can see that the lowest error is usually close to a flow density of $\rho = 2$ and for a flow density of 2, the actual error is below what Equation 6 predicts (shown by the horizontal lines in the plots).

D Taming the multi-resolution beast

The number of parameters in a multi-resolution bitmap is quite large. We now discuss a simple strategy to automatically set these parameters.

Assume for now that we know k . The components of the multi-resolution bitmap have to be accurate for any flow density between ρ_{min} and ρ_{max} . The components are most inaccurate at the ends of the ranges they cover. We can numerically compute ρ_{min} and ρ_{max} by looking for the unique number ρ for which $\max(error(\rho), error(\rho/k))$ is minimized (using Equation 5 for $error(\rho)$).

Once we have the flow densities at the ends of the range, we can compute the size of the bitmap b that guarantees the accuracy A . Equation 4 applies to last component and it gives larger errors than Equation 5 that applies to the others. Therefore, unless we increase the number of bits in the last component, its error will be unacceptably large at ρ_{min} .

For now we assume that we do not change the size of the last component. The largest possible number of flows N will dictate the number of components c . The flow density at component i will be $\rho_i = n/(k^{i-1}b) * (k-1)/k$. The largest number of flows the multi-resolution bitmap can reliably identify is the one reached when component $c-1$ reaches flow density ρ_{max} . Therefore $N \leq (k-1)/k b \rho_{max} k^{c-2}$. From here we obtain $c = 2 + \lceil \log_k(N/(k/(k-1) b \rho_{max})) \rceil$.

⁸The flow density is the ratio between the average number of flows and the number of bits in the virtual bitmap. Since the number of flows in various measurement intervals varies around the average, the flow density also varies.

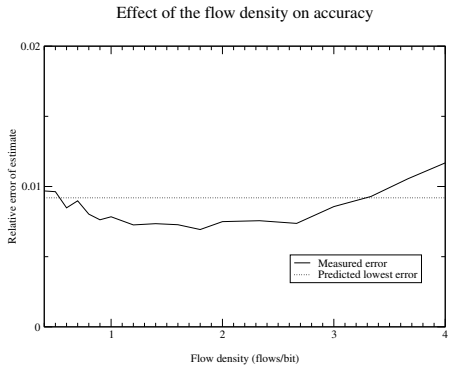


Figure 15: $b=n/4$ (MAG)

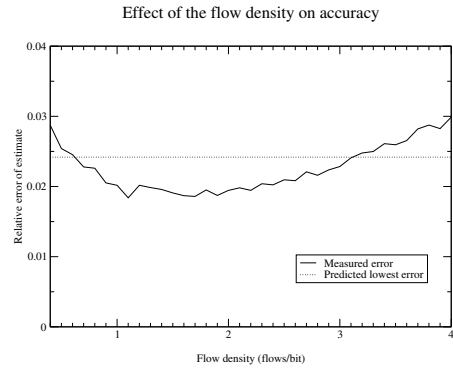


Figure 18: $b=n/4$ (IND)

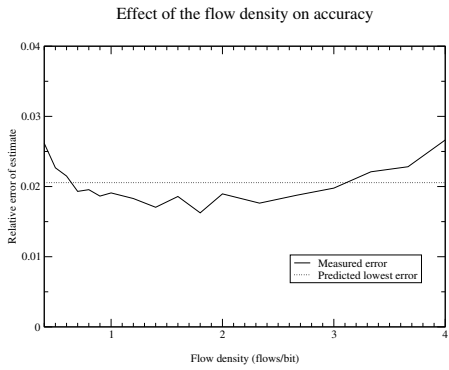


Figure 16: $b=n/20$ (MAG)

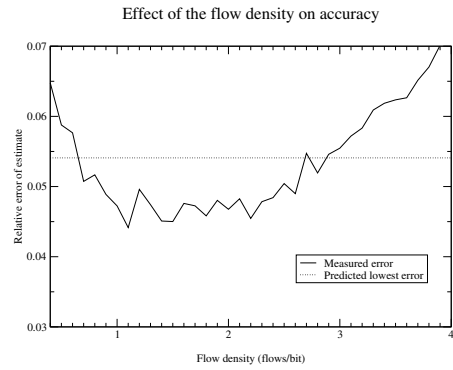


Figure 19: $b=n/20$ (IND)

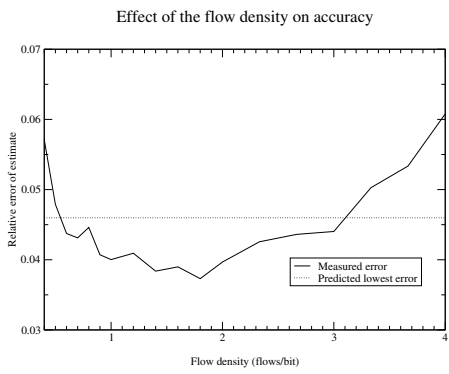


Figure 17: $b=n/100$ (MAG)

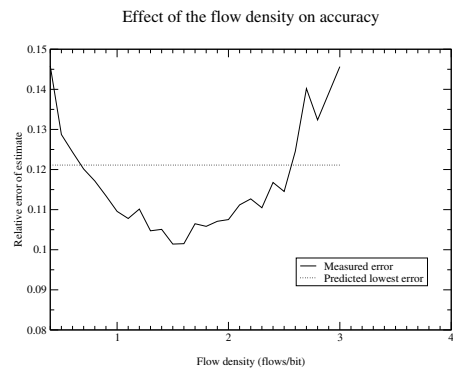


Figure 20: $b=n/100$ (IND)

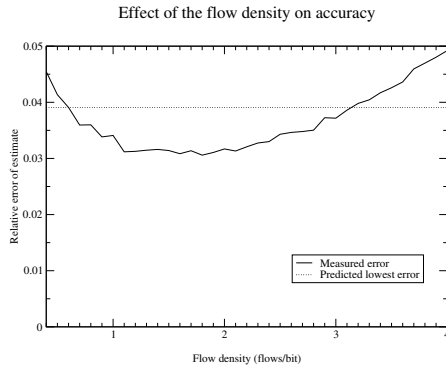


Figure 21: $b=n/4$ (COS)

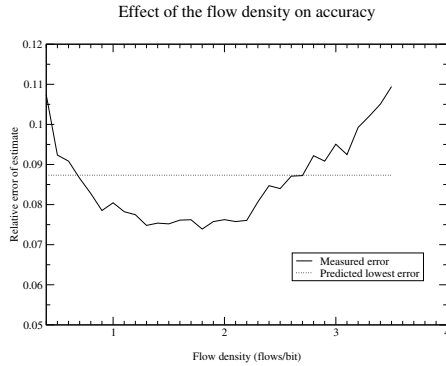


Figure 22: $b=n/20$ (COS)

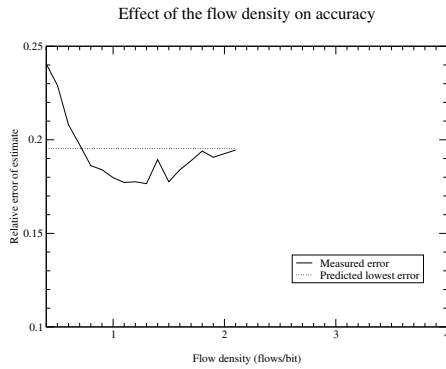


Figure 23: $b=n/100$ (COS)

k	ρ_{min}	ρ_{max}	$k/(k-1) b * A^2$	$k/(k-1) b * A^2(1 - 1/k)$
2	1.6213	3.2426	1.8643	1.344810
3	1.1888	3.5663	2.2623	1.372828
4	0.9540	3.8159	2.6595	1.438817
8	0.5621	4.4966	4.0987	1.724672
16	0.3278	5.2440	6.6287	2.241354
32	0.1881	6.0201	11.1480	3.116098

Figure 24: Comparison between some values of k

So far, we know how to configure and operate the multi-resolution bitmap once we have k . Now we address the problem of choosing the k that minimizes the total amount of memory used. There is a tradeoff between the size of the bitmap and the number of bitmaps and both are determined by k . With large k we need few components to cover all possible values of n , but because ρ_{min} and ρ_{max} are extreme, the accuracy at the boundaries is much worse than with the ideal density, therefore we need large bitmaps. With a smaller k we need more bitmaps, but they can be smaller since they will operate close to the ideal density.

Given N and A , c varies with k roughly as $1/\ln(k)$. Since $B \approx bc$, we can use $b/\ln(k)$ (where b is computed based on k) as an approximative indicator of the amount of memory needed for bitmaps using k .

Table 24 shows numeric values of this indicator for some values of k . The first column is the value of k , the second and third are the ρ_{min} and ρ_{max} that have the ratio k and require the smallest physical bitmap, the next column shows how many times the corresponding bitmap is larger than $1/A^2$ and the last column shows our indicator of the total memory needed by the virtual bitmap. The table shows us that the best value for k is 2. Nearby values are also good, but as we go further, our approximate indicator increases considerably, so it is safe to assume that they will not result in configurations requiring less memory.

There are two more details we need to consider before giving our final algorithm for dimensioning the multi-resolution bitmap. The first is that if the algorithm is implemented in hardware, it simplifies the implementation if k and b are powers of two. The other is that we can “stretch” the last component so that it can be used too. We have to make sure that at the two ends of its range the last component is accurate: the enlarged last component has to be such that by applying Equation 4 we obtain an acceptable error when the flow density is ρ_{min} and respectively when the number of flows is N . If

```

COMPUTECONFIGURATION(N, A)
1  lowestMemorySoFar = ∞
2  for k = 2 to 16
3      b = ⌈(k - 1)/k * errorCoefficient(k)/A2⌉
4      if hardware
5          if k is a not a power of 2
6              skip to next k
7          endif
8          b = (k - 1)/k * 2⌈log2(b)⌉
9      else
10         b = (k - 1) * ⌈b/(k - 1)⌉
11     endif
12     c = 2 + ⌈logk(N/(bρmax(k - 1)))⌉
13     B = b(c - 1) + bk/(k - 1)
14     ρlow = ρmin(k)
15     for lastb = bk/(k - 1) to bk/(k - 1) + b
16         if hardware and lastb is a not a power of 2
17             skip to next lastb
18         endif
19         ρhigh = N/(lastb kc-2)
20         if √((eρlow + ρlow - 1)/(ρlow √lastb)) ≤ A and √((eρhigh + ρhigh - 1)/(ρhigh √lastb)) ≤ A
21             c=c-1
22             B = b(c - 1) + lastb
23             break loop
24         endif
25     endfor
26     if B < lowestMemorySoFar
27         (bestk, bestb, bestc, lowestmemorysofar) = (k, b, c, B)
28     endif
29 endfor
30 return (bestk, bestb, bestc, lowestmemorysofar)

```

Figure 25: Algorithm for computing the best configuration for a multi-resolution bitmap

A / k	b*k/(k-1)	c / B		
		N=1,000,000	N=100,000,000	N=10,000,000,000
10% / 2	188 (256)	12 (12) / 1270 (1664)	19 (18) / 1928 (2432)	25 (25) / 2492 (3328)
10% / 3	228	8 / 1338	12 / 1946	16 / 2554
10% / 4	268 (512)	6 (6) / 1314 (2432)	10 (9) / 2118 (3584)	13 (13) / 2721 (5120)
10% / 8	416 (512)	5 (4) / 1876 (1856)	7 (7) / 2604 (3200)	9 (9) / 3332 (4096)
3% / 2	2072 (4096)	9 (8) / 10922 (18432)	15 (14) / 17138 (30720)	22 (21) / 24390 (45056)
3% / 3	2514	6 / 11458	10 / 18162	14 / 24866
3% / 4	2956 (4096)	5 (4) / 12363 (13312)	8 (8) / 19014 (25600)	11 (11) / 25665 (34816)
3% / 8	4560 (8192)	3 (3) / 12974 (22528)	6 (5) / 24944 (36864)	8 (8) / 32924 (58368)
1% / 2	18644 (32768)	6 (5) / 70315 (98304)	12 (11) / 126247 (196608)	19 (18) / 191501 (311296)
1% / 3	22626	4 / 72948	8 / 133284	12 / 193620
1% / 4	26596 (32768)	3 (3) / 71369 (81920)	6 (6) / 131210 (155648)	10 (10) / 210998 (253952)
1% / 8	40992 (65536)	2 (2) / 81090 (122880)	5 (4) / 188694 (237568)	7 (7) / 260430 (409600)

Figure 26: Configurations for the multi-resolution bitmap

we stretch the last component, we need to change line 18 in the algorithm for computing our estimate of the number of flows (shown in Figure 2) so that it uses the new size of the last component instead of b .

Figure 25 gives the pseudocode of our algorithm for computing the best configuration for the multi-resolution bitmap. The algorithm simply tries all values of k from 2 to 16 (the outer loop from line 2 to 29) and returns the configuration using the least amount of memory (line 30). The function `errorCoefficient` used in line 3 computes the error coefficient that corresponds to k based on Equation 5. It can look it up in a table similar to table 24.

Similarly on line 12 ρ_{max} and on line 14 ρ_{min} can be looked up in a precomputed table. Lines 14 to 25 implement the “stretching” of the last component: we assume we reduce the number of components c by one; if we can find a size for the last component so that the total amount of memory required is less than it was before (ensured by the loop limit on line 15) and it is accurate enough (the if on line 20), then we stretch it (lines 22 and 23).

Table 26 shows the configurations computed by our algorithm that achieve an accuracy of 10%, 3% or 1% for a number of flows up to 1,000,000, 100,000,000 or 10,000,000,000. For brevity we included only values 2,3,4 and 8 for k . The values in parenthesis correspond to the hardware implementation. Not surprisingly, 2 is almost always the best choice for k for the software implementation.⁹ For the hardware implementation $k=2$ is often the best choice, but $k=4$ is better when its required bitmap size for a given accuracy is closer to a power of 2 (e.g. for $A=10\%$ and $A=1\%$).

Ignoring the small differences introduced by one choice or another of k and stretching of the last component we can give a formula that asymptotically describes the memory requirements of a multi-resolution bitmap given the desired accuracy A and the maximum number of flows N . We obtain this by assuming that $k = 2$ is the best choice and assuming that the size of the bitmap is not restricted to powers of 2. By substituting the proper values $k = 2, b = 0.869/A^2$ and $c = 2 + \lceil \log_2(N / (6.485b)) \rceil$ in $B = b(c-1) + bk/(k-1)$ we obtain:

$$B \approx \frac{0.87}{A^2} \lceil \log_2(N) + \log_2(A^2) + 0.5 \rceil \quad (7)$$

⁹For $N=10,000,000,000$ $A=10\%$ and for $N=100,000,000$ $A=3\%$ $k=3$ is slightly better.

E Multi-resolution bitmap versus probabilistic counting

Even though it might seem surprising at first, the data collected by a multi-resolution bitmap with $k = 2$ and no stretching of the last component is isomorphic to the data collected by a probabilistic counting algorithm configured in a certain way (assuming we have good hash functions). The probability of the incoming packet to hash to component i is $1/2^i$ for all components but the last for which it is $1/2^{c-1}$. Each component but the last has b bits.

The probability that the packet hashes to a given bit at component i is $1/2^i * 1/b = 1/(b * 2^i)$ (this also holds for the last component). Therefore, for each i from 1 to $c - 1$ we have b bits that have a probability of $1/(b * 2^i)$ of “catching” the incoming packet plus we have $2 * b$ bits that have the probability $1/(b * 2^c)$ of “catching” the incoming packet. All incoming packets map to exactly one bit.

Probabilistic counting of Flajolet and Martin uses $nmap$ bitmaps of size L . Each bitmap has a probability of $1/nmap$ of “catching” a random database record. Within each bitmap, bit i has a probability of $1/2^i$ of catching the record. The last bit acts as a “catch-all” for all numbers of consecutive zeroes of L or more in the hash, so it has a probability of $1/2^{L-1}$ of catching the packet. Overall for each i from 1 to $L - 2$ we have $nmap$ bits that have a probability of $1/(nmap * 2^i)$ of “catching” the record plus we have $2 * nmap$ bits that have a probability of $1/(nmap * 2^{L-1})$ of “catching” the record.

We can see in Figure 27 that when $b = nmap$ and $c = L - 1$ the two algorithms have the same number of bits and the probability distribution of bits getting set is the same. Therefore we conclude that the data collected by the two algorithms is equivalent. What is the difference then? The most important difference is the way the collected data is interpreted. As both analysis and experiments show this leads to our algorithm being more accurate when the number of flows is small. Another difference is that we have different rules for configuring the algorithm which, as experiments show, result in somewhat more accurate estimates when the number of flows is large.

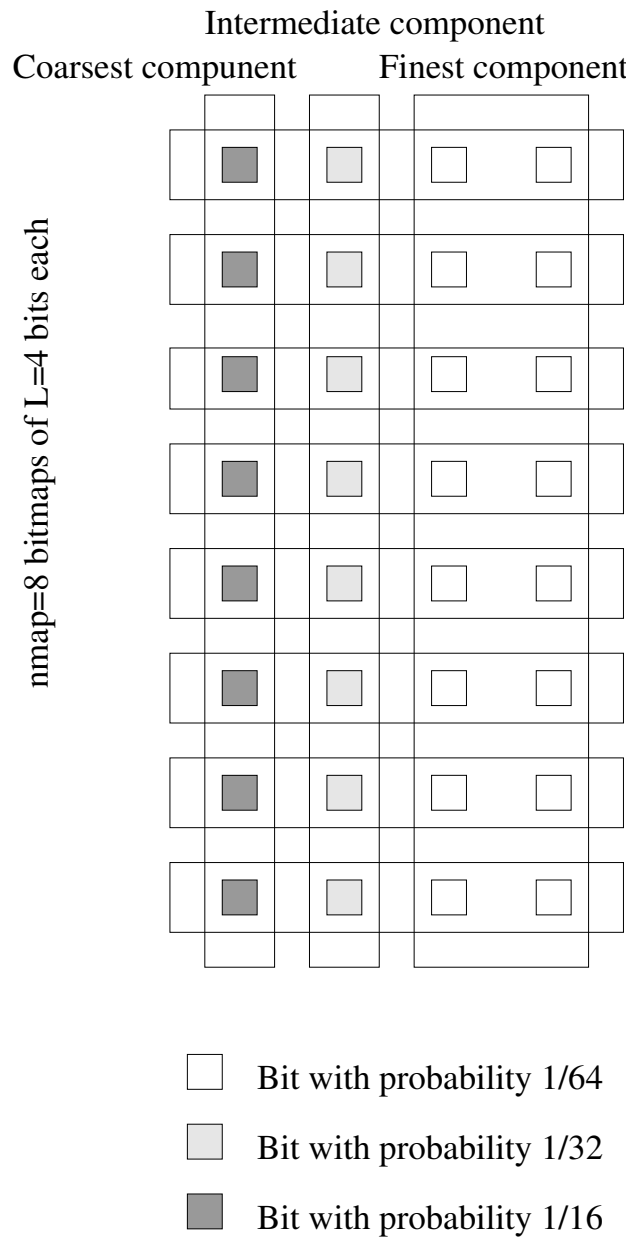


Figure 27: Probabilistic counting groups the bits horizontally into bitmaps that contain bits with different probabilities of being set, while multiresolution bitmaps group bits vertically with bits with the same probability of being set in the same component