A Robust System for Accurate Real-time Summaries of Internet Traffic

Ken Keys
CAIDA
San Diego Supercomputer Center
University of California, San Diego
kkeys@caida.org

David Moore
CAIDA
San Diego Supercomputer Center
University of California, San Diego
dmoore@caida.org

Cristian Estan
Computer Sciences Department
University of Wisconsin-Madison
estan@cs.wisc.edu

ABSTRACT

Good performance under extreme workloads and isolation between the resource consumption of concurrent jobs are perennial design goals of computer systems ranging from multitasking servers to network routers. In this paper we present a specialized system that computes multiple summaries of IP traffic in real time and achieves robustness and isolation between tasks in a novel way: by automatically adapting the parameters of the summarization algorithms. In traditional systems, anomalous network behavior such as denial of service attacks or worms can overwhelm the memory or CPU, making the system produce meaningless results exactly when measurement is needed most. In contrast, our measurement system reacts by gracefully degrading the accuracy of the affected summaries.

The types of summaries we compute are widely used by network administrators monitoring the workloads of their networks: the ports sending the most traffic, the IP addresses sending or receiving the most traffic or opening the most connections, etc. We evaluate and compare many existing algorithmic solutions for computing these summaries, as well as two new solutions we propose here: "flow sample and hold" and "Bloom filter tuple set counting". Compared to previous solutions, these new solutions offer better memory versus accuracy tradeoffs and have more predictable resource consumption. Finally, we evaluate the actual implementation of a complete system that combines the best of these algorithms.

Categories and Subject Descriptors

C.2.3 [**Computer Communication Networks**]: Network Operations—*Network monitoring*

General Terms

Measurement, Experimentation, Performance, Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMETRICS'05, June 6–10, 2005, Banff, Alberta, Canada.
Copyright 2005 ACM 1-59593-022-1/05/0006 ...$5.00.

Keywords

Passive monitoring, measurement, sampling, traffic estimation, adaptive response

1. INTRODUCTION

In order to run networks efficiently, network administrators need to have a good understanding of how their networks are used and misused. Thus, an important tool in each network administrator's toolbox is the ability to monitor the traffic mix, especially on important links.

Increases in traffic volumes made possible by increases in line speeds are one of the main driving forces behind the evolution of traffic measurement. While capturing traces of packets or packet headers is feasible at low speeds, at higher speeds aggregation is necessary to reduce the amount of traffic measurement data. Most routers report traffic measurement data in the NetFlow format [26] that aggregates all packets belonging to the same flow into a flow record. This kind of aggregation at the router makes traffic measurement more fragile, because the assumptions on which aggregation is based do not hold in some traffic mixes. When small flows predominate, NetFlow aggregation does not help at all. Anecdotal evidence abounds about routers using NetFlow crashing because a denial of service attack with randomly faked source addresses makes them run out of memory. NetFlow uses packet sampling at the router [27] to reduce processing memory usage and the size of its output, but the aggressive sampling needed to keep resource consumption under control under the most extreme traffic mixes compromises the accuracy of the measurement results.

This paper presents a robust traffic measurement system that handles unfriendly traffic mixes by gracefully degrading the accuracy its results. Furthermore, the measurement results it produces are concise traffic summaries and their accuracy is as high as possible with the resources available. The structure of the paper is as follows. In Section 2 we describe the goals of our traffic measurement system. In Section 3 we discuss related work, including prior algorithms we incorporated into our system. In Section 4 we describe the algorithms for computing the summaries, including our two new algorithms: Bloom filter tuple set in Section 4.1.2 and flow sample and hold in Section 4.2.2. In Section 5 we describe our system as a whole, including the adaptation methods we use to achieve robustness with respect to memory and CPU usage. In Section 6 we measure the performance of different configurations of our system on real traces. We also evaluate how well they perform in response to adverse network traffic. We conclude with Section 7.

2. MEASUREMENT SYSTEM GOALS

We set out to build a system that produces compact and timely traffic summaries. Section 2.1 describes the summaries we are interested in. In section 2.2, we identify four potential bottlenecks and present the less tangible goals we set for our system such as graceful degradation of the accuracy of summaries when faced with unfriendly traffic mixes and isolation between the resource consumption of the algorithms generating summaries.

2.1 Traffic Summaries

The traffic summaries we want our system to produce can be divided into two categories: global traffic counters and "hog" reports, which list heavy hitters by packets, bytes and flows. Both these types of summaries are implemented by current systems [25, 1] and widely used by network administrators. However, computing accurate and timely hog reports for high speed links is challenging. These summaries reflect the traffic of the monitored link over fixed duration measurement intervals.

Our global counters measure the counts of the following entities in each interval: packets sent, bytes sent, active flows, active source IP addresses, active destination IP addresses, active protocol/source port pairs, and active protocol/destination port pairs. Since some of these numbers cannot be measured with simple counters, we need more complicated "flow counting" algorithms.

We will refer to sources or destinations that send or receive many packets or bytes as "packet hogs" or "byte hogs". Sources or destinations that have many flows are "flow hogs". The measurement system produces four types of hog reports keyed by specific packet header fields: source IP, destination IP, source port and protocol, and destination port and protocol. The system produces byte, packet and flow hog reports for each key. The system thus produces a total of 12 hog reports. For example, we will have a report that lists the source IP packet hogs, and how many packets each of those sources sent.

In particular, these reports allow answering of many common questions asked by network operators. For example, the port-based reports provide information about application usage. Hosts which are engaged in malicious activity are often visible on source IP flow hog reports, since port scanning or spam relaying generates many outbound flows. Because the total volume of this malicious traffic is low, these hosts are not typically visible on packet hog or byte hog reports.

Some network administrators may be interested in summaries other than the ones presented in this paper. For example, one might want to aggregate traffic by longest matching prefix or AS number instead of IP address. Or one might want to measure the out-degree of source IP addresses (number of destination addresses to which they connect) or in-degree of destination addresses instead of flow counts. The methods we use could be readily applied to those measurements. Furthermore, with a software based architecture such as ours, these changes are relatively easy to implement.

2.2 Robustness and Isolation

The underlying architecture of our system is a general purpose computer with an OC-48(2.5Gbps) DAG capture card [30], as shown in Figure 1. However, we expect most of our techniques to prove useful on different architectures that have some of the same potential bottlenecks, for example devices built around a network processor. We identify four potential bottlenecks: memory, CPU processing power, bus bandwidth, and output network speed.

The simplest bottleneck for the system to avoid is output bandwidth. By exporting only the compact summaries described above, we minimize the amount of data to be sent over the network. Next,

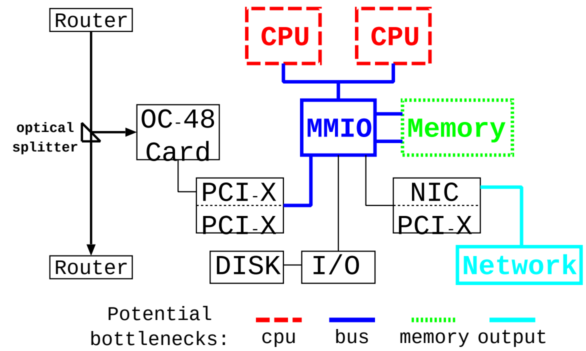

Figure 1: The underlying architecture of our system on a current high-end PC motherboard [28, 20, 21]. Potential bottlenecks for measurement exist in CPU processing, bus bandwidth, memory size, and output network speed. Even in a specialized hardware deployment, such as on a router blade, these same potential bottlenecks would exist.

we address the potential memory bottleneck by using memory-efficient counting algorithms and accurate sampling techniques.

We present implementation techniques that reduce the CPU usage, but the CPU can still be overwhelmed by a large number of packets, as can the system bus. As a last resort, we can protect against this problem by performing some kind of sampling like Sampled NetFlow [27] or packet selection [2] directly on the card. Since the card we use does not currently support this but is likely to in the future, we use simulations to evaluate how adaptive sampling can provide robustness with respect with CPU usage.

Our system can overcome the potential bottlenecks because we allow its results to be approximate. However, we want to achieve the most accurate results possible with the existing resources. When faced with atypical traffic that strains some system resources, the system should continue providing summaries, though possibly at a lower accuracy. For example, a distributed denial of service (DDoS) attack with fake source addresses might exhaust the available memory by consuming too many entries in a source IP table. The system could react by refusing to create new entries in that table, but this is a poor response because big hogs that do not appear until after this decision is made will be completely omitted from the table. Small errors in the counters and the misordering of similar entries in the hog reports are acceptable ways of degrading accuracy, but omitting a source whose traffic is significantly above that of others included in the hog report is not.

Sharing memory and CPU resources between the components computing the various summaries can result in cheaper and more efficient systems when compared to the alternative of complete isolation between the components. However, we do not want a traffic mix that strains one of the components to starve the others of resources. For example, it is acceptable that in response to a DDoS attack the system decreases the accuracy of the source IP report, but the destination IP report (which would not be affected by the attack if implemented in isolation) should not be affected.

3. RELATED WORK

NetFlow [26] is a widely deployed general purpose measurement feature of Cisco and Juniper routers. The volume of data produced by NetFlow is a problem in itself [17, 10]. To handle the volume and traffic diversity of high speed backbone links, NetFlow resorts to packet sampling [27]. The sampling rate is a configuration parameter set manually and seldom adjusted. Setting it too low causes inaccurate measurement results; too high can result in the measurement module using too much memory and processing power, especially when faced with increased or unusual traffic, which can lead to dropping data and thus very poor accuracy. Recently, Adaptive NetFlow [13] has been proposed to adaptively tune the sampling rate to memory consumption. The advantage of flow records over traffic summaries computed at the measurement device is that they can be used for a wide variety of analyses after they reach the remote collection station. However, when we know in advance what aggregations we want, we can produce traffic summaries with less error. The Gigascope project [8] exemplifies an approach that maintains both the accuracy one can achieve by processing raw data locally and the flexibility of a general SQL-like query language, but without any guarantee of robustness to unfavorable traffic mixes.

The two algorithmic problems we need to solve to compute accurate traffic summaries are (1) identifying and measuring in a streaming fashion all addresses and ports with heavy traffic, and (2) counting the number of flows for each. While existing solutions address these problems for a subset of the measurements, none provides the whole picture. Identifying heavy hitters has been addressed in both database [19, 16, 7] and networking [14] contexts. Counting the number of distinct items has been addressed [18, 31, 3, 12] by the database community. The related problem of finding the flow hogs and counting their flows have can be easily solved by using hash tables that explicitly store all flow identifiers [23, 25], but the memory cost can be excessive. This problem is algorithmically equivalent to the problem of finding superspreaders which has been addressed by Venkataraman et al.[29] and they have independently proposed techniques equivalent to our flow sample and hold and Bloom filter tuple set counting. Efficient bitmap algorithms for flow counting have been proposed [15]. Our system directly uses some of them while improving on others. Bloom filters [4] are a very useful data structure for testing set membership; in Section 4.1.2 we use them to count flows. Kumar et al.[24] have used a variation of the Bloom filter for the related but different problem of counting packets in a flow.

4. ALGORITHMS FOR TRAFFIC REPORTS

The "global counters" that are part of the reports do not pose significant issues: for the bytes and packets we use simple counters, and for counting distinct addresses, ports, and flows we use multiresolution bitmaps[15]. Since these have low and constant CPU and memory usage and well understood accuracy, the rest of the paper will focus on the hog reports. A simple way of producing a hog report for a given key is to keep a hash table with a counter for each key in the traffic, and just report the top entries at the end of the measurement interval. This approach has two problems: flows can not be counted with simple counters, and the tables can get too large. Section 4.1 discusses the algorithms we use for counting flows and Section 4.2 those for identifying the entries worth keeping in the tables.

4.1 Flow counting

Counting flows for each table entry is harder than counting bytes or packets. We must distinguish between packets belonging to old and new flows, and increment the flow counter only if the flow is new. This is true for the global flow counter as well as flow counters for individual table entries. After discussing existing algorithms for counting flows, we propose two new ones, Bloom filter tuple set counting and list-triggered bitmaps.

4.1.1 Background

An obvious method of counting flows (and bytes and packets) by source address, destination address, source port and destination port is to maintain a global "tuple table" keyed by flow ID and update it for each packet. At the end of each measurement interval, we aggregate the entries into the four target tables by the appropriate keys. This algorithm is used by CoralReef's [23] [25] `crl_flow`. It gives exact counts, but the tuple table uses a large amount of memory, and much of the processing cost is concentrated at the end of the interval. Under extreme conditions, such as a worm attempting to spread or a DDoS attack, the tuple table overflows exactly when we are most interested in the results.

We can also estimate the number of active flows without explicitly storing all flow identifiers. Start with an empty bitmap, set the bit in the bitmap corresponding to the hash value of the flow ID of each packet, and at the end of the interval estimate the number of active flows based on the number of bits set. This algorithm, called linear counting [31] or direct bitmap [15], provides accurate estimates but its memory requirements scale almost linearly with the maximum number of active flows. The size of the bitmap also depends on the required accuracy of the estimate. Similar algorithms such as multiresolution bitmaps [15] and probabilistic counting [18] use more complex mappings from flow IDs to bits and their memory requirements scale logarithmically with the maximum number of active flows. With a few kilobytes these algorithms can give estimates with average errors of around 3% for up to hundreds of millions of flows.

4.1.2 Bloom Filter Tuple Set

With the tuple table algorithm, we can spread out the expensive end-of-interval aggregation by maintaining the four target tables during the interval, incrementing the flow counter in the corresponding entry of each whenever we add a new tuple table entry. However this does not reduce the memory usage of the tuple table.

Note that the tuple table is now used only to decide whether a packet belongs to a new flow. We can think of this as using the tuple table to test whether the flow ID of a packet is in the set of flow IDs already seen. If we are willing to accept an estimate of the flow count for each entry, we can replace the tuple set with a more compact fixed size structure designed for testing set membership: a Bloom filter [4]. A Bloom filter is implemented as a bitmap of b bits, initially empty, and k independent uniform hash functions with range $\{0...b-1\}$. To insert a flow ID into the set, we compute all k hash functions on the flow ID and set the bits corresponding to each hash value. To test whether a flow ID is new, we test the bits corresponding to each hash of the flow ID. If all the bits are set, we assume the flow ID is not new; if any of the bits are 0, the flow ID is definitely new and we increment the counters in the target tables. If we have seen n flows so far, the probability of a false positive, i.e. mistakenly concluding we have seen a flow before when we actually have not, is $(1-(1-1/b)^{kn})^k \approx (1-e^{-kn/b})^k$.

Even if the Bloom filter falsely indicates that we have seen a particular flow ID before, we can still identify it as new if it has a source address, destination address, source port or destination port we have not seen before. Since we must already check these keys for insertion in the corresponding tables, we have four additional checks for newness with no additional cost, greatly reducing the effect of false positives from the Bloom filter. It is difficult to predict theoretically how much this helps, but in practice a large fraction of the entries in each table have very few flows, so most Bloom false positives among that fraction will still be counted. For example, consider a worm attacking a large number of destinations. For many of those destinations the attack flow will be their only flow, so many of the Bloom filter false positives are still counted, keeping the attacker's flow count relatively accurate. Similar logic holds for DDoS attacks with a large number of spoofed source addresses.

As described, the Bloom filter tuple set algorithm always gives lower bounds for flow counts, because the Bloom filter test and table insertion tests will never incorrectly identify an existing flow as a new flow. In many measurement contexts, it is useful to know that the estimate is a lower bound.

4.1.3 List-Triggered Bitmaps

Instead of a global tuple table or Bloom filter and a simple flow counter in each entry of the target tables, we can add a multiresolution bitmap (much smaller than the Bloom filter) to each entry. In typical traffic mixes most of the IP sources and destinations have very few flows, so per-entry flow counters do not need as much memory as multiresolution bitmaps configured to work for up to hundreds of millions of flows. The triggered bitmap algorithm [15] saves memory by starting with only a small direct bitmap in each new entry and allocating a multiresolution bitmap only when the number of bits set in the direct bitmap exceeds a trigger value. To avoid bias, the multiresolution bitmap is updated only for packets that hash to bits not set in the direct bitmap. The direct bitmap is not very accurate because it is small, and the multiresolution bitmap loses accuracy because it only covers a sample of what it would cover alone.

We propose an alternative to the triggered bitmap that avoids loss of accuracy in its multiresolution bitmap while using comparable amounts of memory: list-triggered bitmaps. We replace the direct bitmap with a small list of up to g flow identifiers. The value of g is small, typically between 2 and 8, so this list is most efficiently implemented as an array. To further save space we do not store flow IDs but 64 bit hashes of flow IDs. For each packet, we append the hash value of the flow ID to the list if it was not already in the list. If the list is full when we try to append a new value, we allocate a multiresolution bitmap and insert the new hash value and all the old hash values from the list into it. When the true number of flows n is less than or equal to the maximum list size g, the multiresolution bitmap is never allocated, and the estimate is exactly the number of hash values in the list; the only source of error is collisions in the hash function, which is negligible for the values of g we use. Otherwise, we use the multiresolution bitmap algorithm, with one refinement: if the multiresolution estimate is less than $g + 1$, we say the list-triggered estimate is $g + 1$, because the multiresolution bitmap would not have been allocated unless there were at least that many hash values.

4.2 Identifying important entries

For a lightly loaded OC-48 with a favorable traffic mix, a measurement system with a few hundred megabytes of memory and efficient algorithms for counting flows can afford to keep an entry for each source and destination IP. However, under adverse traffic mixes such as massive DoS attacks with source addresses faked at random or worms aggressively probing random destinations, keeping even a small entry for each unique IP address can consume too much memory for even a generously endowed workstation. Thus while we want to keep state for the hogs, we cannot afford to keep state for all entries. We need algorithms to identify the hogs.

4.2.1 Packet Sample and Hold

The sample and hold algorithm [14] can be used to identify and accurately measure the packet hogs without keeping state for most other entries. Packets are sampled at random; for each sampled packet, an entry is created in the target table if one does not already exist, and all packets corresponding to that entry are counted from then on. We will refer to this algorithm as "packet sample and hold" or "PSH", to distinguish it from another algorithm we introduce later. The sampling probability is a "tuning knob" we can use to trade memory for accuracy: a high probability gives accurate results, a lower one reduces memory usage but allows more packets go uncounted before the entry is created.

The analysis of PSH [14] shows that it identifies the packet hogs with high probability. Naive thinking suggests that PSH could also be used to identify flow hogs, since a source can have many flows only if it has at least as many packets in the traffic, but as the following example shows, PSH cannot always achieve this goal.

Consider a 5 minute interval of traffic on a saturated OC-48 containing 60,000 flows, each made up of 1,000 packets of 1,500 bytes each (say, 60,000 pairs of distinct hosts doing peer-to-peer sharing of 1.5 MB files). Add to this a port scanner that sends a single 40-byte scan packet to each of 50 destinations. Of course we want to detect the port scanner, because it the largest flow hog. The measurement system must assume that each packet might have a different source IP, so if the source IP table has space for only 50,000 entries, then with PSH we can sample at most one packet in 1,200 without risking filling up the table. At this sampling rate the probability of catching the port scanner is approximately 1/24. This is so low because the probability of a source IP getting an entry in the table depends on the number of packets it sends, not on the number of flows it has. In general, traffic mixes dominated by very many sources with few high-packet flows will make it hard for PSH to detect early enough the sources that have many low-packet flows. While the traffic mix from this example is not typical, this failure mode is unacceptable; we want a system that is robust in the face anomalous traffic.

4.2.2 Flow sample and hold

To accurately identify the flow hogs regardless of how many packets they have, we introduce flow sample and hold (FSH). It is similar to PSH but its sampling function favors entries with many flows. We hash the flow identifier of every packet; if the hash value is less than the control variable f (where f is in the range of hash values), we create an entry in the target table. FSH with a 32-bit hash function has a flow sampling probability of $f/2^{32}$. Note that all packets of a flow have the same hash value, so the number of packets in a flow does not affect its probability of triggering the creation of an entry for its source IP. Furthermore, as the number of flows a source has increases, the probability of it not getting an entry decreases exponentially. Therefore big flow hogs will get entries early, and the flow counter in those entries will count all corresponding flows that are active after that point.

More formally, we can quantify the probability that a source sending a certain number of flows evades detection. Let F be the number of flows of a given source and $p_1 = f/2^{32}$ be the flow sampling probability. The probability that this particular source will not have an entry is $p_{miss} = (1 - p_1)^F \approx e^{-Fp_1}$. We can also bound the expected number of a source's flows that we miss before creating an entry for the source, which is equal to the absolute error in the entry's flow count estimate, since we accurately count all flows after the entry is created. Thus $E[missed] = \sum_{i=0}^{F} i \cdot p(\text{miss exactly } i \text{ flows}) = \sum_{i=0}^{F} i \cdot p_1(1-p_1)^i \leq \sum_{i=0}^{\infty} i \cdot p_1(1-p_1)^i = \ldots = 1/p_1 - 1$. For the example from the previous section, let's say we use FSH with a sampling probability of 0.1. The probability that the port scanner gets an entry is $(1 - 0.9^{50}) \approx 99.5\%$. At the same time the sources of peer-to-peer traffic that have one flow each will have a probability of 0.1 of getting an entry. Thus the peer-to-peer traffic will add only around 60,000*0.1= 6,000 entries to the source IP table. So, FSH does a much better job than PSH of finding flow hogs while reducing the number of "noise" entries with low flow counts.

Can FSH replace PSH? Is it guaranteed to catch packet hogs (or byte hogs)? The answer is no. Extending the example above, imagine that we also have a host sending only one 100 MB file through a single TCP connection. This is by far the largest sender, but there is a 90% probability that FSH will not sample its single flow, and thus ignore this important source. The obvious answer for a system that aims to detect both packet hogs and flow hogs is to use both PSH and FSH to populate the tables with entries.

The question that arises naturally is whether we also need a "byte sample and hold" (BSH) algorithm to detect the byte hogs. The reason we need both PSH and FSH is that the ratio between the number of packets in two flows can be in the thousands. However, the ratio between the number of bytes in two packets does not exceed 38 in current traffic mixes, as packets range in size from 40 to 1500 bytes. While byte-based sampling as in [14] does give more accurate results than PSH (which samples packets with equal probability), the difference is small, since the ratio between packet sizes is bounded by a small constant and the number of entries we can keep in the tables is much larger than the number of entries we report. An implementation of the system could trivially make BSH available and optional, and let the user decide whether the extra accuracy in catching byte hogs is worth the extra CPU overhead. For simplicity and efficiency, in the rest of the paper we rely on PSH to identify the byte hogs.

5. SYSTEM DESCRIPTION

In this section we describe the actual measurement system we implemented. We first describe in Section 5.1 how a component that computes one hog report can adapt to the traffic mix to avoid exhausting the memory or the CPU. In Section 5.2 we describe how we integrate several such components to form the full system, and show how components make better use of resources by sharing them. In Section 5.3 we describe how we ensure that, despite sharing, the components cannot starve each other of resources.

5.1 Robustness and adaptation

We now discuss how a component that computes a hog report can be robust with respect to memory and CPU usage when faced with adverse traffic. For simplicity, we focus on the packet hog report for source addresses; it is easily generalized to the other 11 hog reports, and we will point out the differences where they matter.

The basic idea is to keep a table with an entry for each source and count the packets they send. Using PSH limits the number of entries created, but we need to choose a good sampling rate: if too high, we run out of memory; if too low, the results will be needlessly inaccurate. Our solution is to adaptively decrease the PSH sampling rate based on how quickly memory is filling up. We start each measurement interval with a sampling probability of 1, creating an entry for each distinct source IP in the traffic stream. If this allocates entries too quickly, we decrease the probability to a value that we estimate will keep the table size within the memory budget until the end of the interval. Appendix A describes exactly how we choose the new probability. When using the Bloom filter tuple set algorithm to count flows on each table entry, the size of each table entry is fixed, so using adaptive sample and hold to control the number of entries is sufficient to control memory use. But when using triggered or list triggered bitmap, entries grow when the number of flows triggers the allocation of a large multiresolution bitmap, so sample and hold is less effective at controlling memory usage.

Decreasing the PSH sampling probability will reduce CPU usage somewhat, but since we still must do a table lookup for every packet, it may not be enough. This is more of an issue for a software implementation than an implementation on a router with specialized hardware. At OC-48 speeds the CPU has an average of 128 ns to process a 40 byte packet. Packet buffers can absorb bursts of packets, but not long streams of back to back short packets that can come for example from a massive DDoS attack. We control CPU usage by adaptively pre-sampling the packets on the capture card before they reach the CPU. See the technical report [22] for details on how we adapt the packet pre-sampling probability. We compensate for the pre-sampling when updating the counters. For example, if the pre-sampling probability is 1/5, each sampled packet is counted as 5 packets. The randomness of pre-sampling reduces counter accuracy, but for packet hogs the relative error is small.

Pre-sampling also reduces the number of table entries, since many of the sources with few packets will have all their packets dropped, suggesting we could use it instead of PSH to control memory usage. However, analysis [14] shows that PSH gives much more accurate results than pre-sampling with the same sampling probability (and thus the same memory usage).

For the byte counters, we can compensate for the pre-sampling the same way we do for packets: we increment the counters by the size of the packet divided by the pre-sampling probability. The situation is more difficult for flow counters. It has been shown that for any flow estimator based on a random sampling of the packets belonging to a source IP, there is a traffic mix for which the estimate is far off from the actual count[6]. Using additional information such as TCP SYN flags in the pre-sampled packet headers can lead to more consistently accurate estimates [11]. These methods apply only to TCP and rely on the end hosts correctly setting the flags. Given that the problem is fundamentally hard and the pre-sampling probability keeps changing to respond to changes in the traffic mix, we adopt a simple solution for the flow counters: we do not compensate for the pre-sampling. While this does result in underestimating the number of flows when the pre-sampling rate is low and the mix contains short flows, we at least know that our flow estimates are a lower bound on the actual flow counts.

Because of the inaccuracies introduced by pre-sampling, our system relies on sample-and-hold to control memory usage, and only lowers the pre-sampling rate below 1 when necessary to control CPU usage. Note that many existing systems do the equivalent of fixed-rate packet pre-sampling all the time, so the effect on them of the problems described above is much worse.

5.2 Structure of the system

When we integrate the modules that compute the various summaries, sharing between them often saves memory and CPU cycles allowing us to maintain higher sampling rates and thus achieve better accuracy. Note that there are three reports for each of the four key types: byte hogs, packet hogs, and flow hogs. Instead of keeping three separate tables for each key type we can use a single table for each, with each entry having all three counters. This makes the entries larger, but since many keys would appear in two or even all three of the reports, having them share a single table entry results in a net reduction of memory use. A clear win of merged tables is that we have to perform only four table lookups per packet instead of twelve, significantly reducing CPU usage. Figure 2 shows how a combined table operates.

We choose the Bloom filter tuple set algorithm for counting flows. One reason is that our experiments from Section 6 show that for most configurations its accuracy is equal to or better than that of any of the bitmap algorithms. The other reason is that we can better control its memory usage because we need to control only the number of entries in the tables. Sharing a single Bloom filter among all four tables is a major reduction in memory and CPU usage. Furthermore, because we use one Bloom filter instead of four, it can be

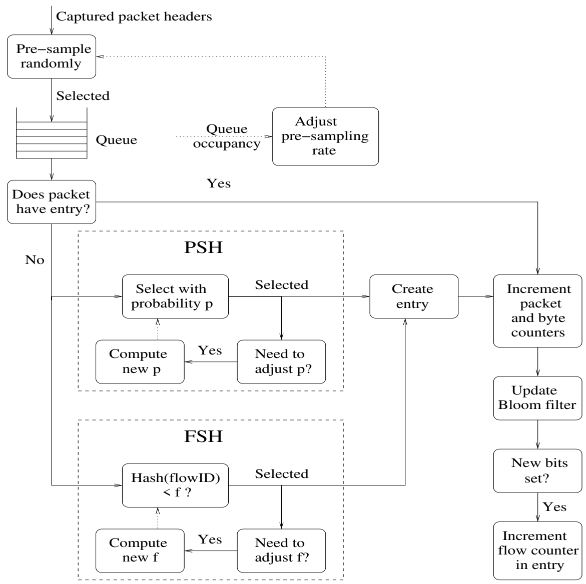

Figure 2: Tables keyed on various fields combine the algorithms for selecting entries (PSH and FSH) with the Bloom filter tuple set algorithm for counting flows. Dynamically adapting the PSH and FSH sampling rates ensures that the system never tries to allocate more entries than memory can hold.

larger, so the flow counters are actually more accurate because this larger shared Bloom filter has fewer errors due to collisions.

The summaries produced by our system also include various global counters (total distinct source IPs, flows, etc.). The total flow count can be estimated by treating the tuple set Bloom filter as a very large direct bitmap generalized to multiple hash functions, so no additional memory is needed. The other global counters are implemented with multiresolution bitmaps. We use an efficient implementation of the H3 hash function family[5], which has the useful property that it can be computed piecewise. When calculating the hash value for the global flow counter, we can save CPU time by reusing the hash values already calculated for the source IP and destination IP counters, since their keys are subsets of the flow key.

For our estimates to be accurate, the hash functions for the Bloom filter and bitmaps must produce uniformly distributed values for any input. H3 hash functions satisfy this requirement. Additionally, we want the hash function to be unpredictable to an external observer so it is impossible to maliciously craft traffic that will subvert our system by causing hash collisions in the Bloom filter, bitmaps, or hash tables, as described by Crosby and Wallach[9]. We achieve this by randomly generating H3 functions.

5.3 Isolation

Isolating the resource consumption of various components of the system ensures that a strain on one component does not hurt the accuracy of the others. Memory is dynamically allocated only by the four tables used to compute the hog reports. Isolating their memory consumption is easy: we divide among them the number of entries we can allocate. Thus while a DoS attack with faked source IP addresses will strain the source IP table and cause a decrease in its PSH and FSH sampling probabilities leading to reduced accuracy of its hog reports, the other tables will be unaffected and the other reports will not lose accuracy. Our current system divides the memory equally among the tables, but this can be overridden through configuration. There is a simple improvement over this strategy, which we have not yet implemented: if some of the tables do not use up their share of the memory (e.g. the port tables), in the next interval we can redistribute the surplus among the others.

We have two algorithms, PSH and FSH, adding table entries. We protect them from each other by dividing the memory budget of each table equally between the two algorithms and adjusting the sampling probabilities of PSH and FSH independently (see Figure 2). If both algorithms sample a packet that causes the creation of a new entry, they each get charged for half an entry.

The packet processing of the modules producing the various summaries is severely intertwined, thus performing packet pre-sampling to separately control the CPU usage of each module is impractical. Fortunately it is also unnecessary for two reasons. Firstly, the average per packet CPU usage of each of the modules is a constant share of the total CPU usage, so no module can take a disproportionate share of the CPU. Therefore all modules need to reduce their CPU usage at the same time: when there are too many packet headers to process. Secondly, for packet pre-sampling to most effectively protect the CPU and the bus it must be implemented on the capture card[1].

6. MEASUREMENT RESULTS

In this section, we use experiments to evaluate various system configurations and validate our design. We present our experimental setup and then describe the experiments to test specific system components. We first compare the two algorithms for identifying important entries, PSH and FSH, to see whether each has its own advantages as predicted by the theoretical analysis. Second, we compare the three flow counting algorithms: triggered bitmap, list-triggered bitmap and Bloom filter tuple set. Third, we investigate the behavior of our adaptation methods and test whether they provide robustness and isolation. Finally, we test the fully configured system with a variety of data sets and PRNG seeds to show that it behaves consistently.

6.1 Experimental setup

We first present our metrics and experimental datasets. The metrics we use to evaluate our system are:

- *overall memory usage* - how much memory was used by the entire process over the 5 minute interval
- *CPU run time* - user and system CPU time usage
- *top N selection error* - measures how far the system was from correctly selecting the top N entries (by packets, bytes, or flows, in a given table). The exact definition and detailed results of this metric are given in the technical report version of this paper [22].
- *RMS relative error* - measures the average error for all of the byte, packet, or flow estimates in a table, giving more weight to values with greater average error:

$$\text{RMS rel. err.} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\hat{n}_i - n_i}{n_i}\right)^2}$$

- *normalized absolute error* - measures the absolute error for all of the byte, packet, or flow estimates in a given table, thus

[1]Since the capture card used in our experiments does not support random packet sampling, we can only simulate pre-sampling.

Dataset	Start Time (UTC)	Duration	Packets	(pkts/sec)	Bytes	(bits/sec)	Flows	(flows/sec)
Robustness testing:								
Backbone	Wed Aug 14, 2002 16:00	5 min.	22.5 M	(74.9 k)	12.8 G	(342. M)	1.21 M	(4.03 k)
Backbone + DDoS	Wed Aug 14, 2002 16:00	5 min.	32.5 M	(108. k)	13.2 G	(352. M)	11.2 M	(37.3 k)
Validation:			Averages of 5 min. samples					
OC48-A	Wed Aug 14, 2002 16:00	1 hour	22.8 M	(76.2 k)	13.1 G	(349. M)	1.22 M	(4.08 k)
OC48-B	Wed Aug 14, 2002 16:00	1 hour	34.9 M	(116. k)	22.8 G	(608. M)	2.62 M	(8.74 k)
Campus	Fri May 16, 2003 16:48	1 hour	17.5 M	(58.5 k)	11.1 G	(297. M)	0.59 M	(1.98 k)

Table 1: Datasets used in study. The "Backbone" test dataset consists of the first 5 minutes of the OC48-A trace. The "Backbone + DDoS" dataset adds 10 million simulated DDoS packets spread over the 5 minutes.

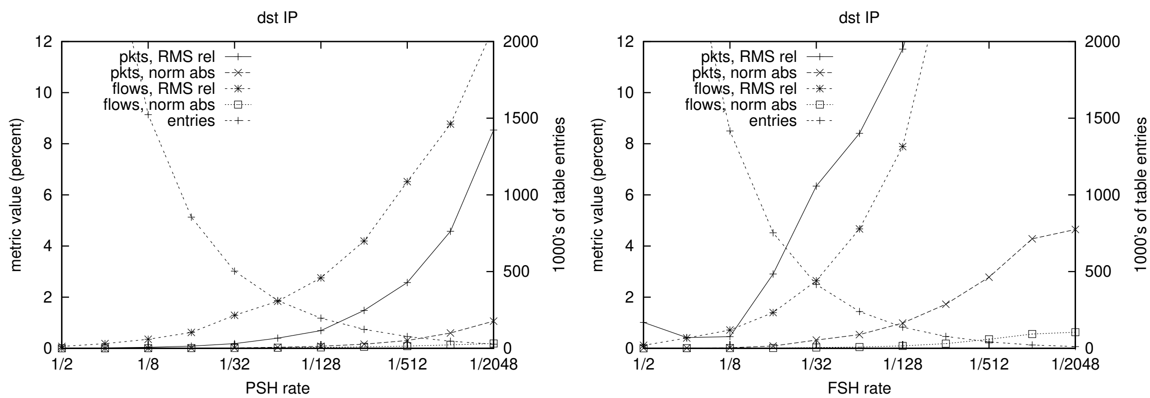

Figure 3: Trading memory for accuracy with sample-and-hold rates. Counting done on Backbone with DDoS with tuple set algorithm.

giving more weight to more important (larger) entries, and normalizing by the sum of the entries:

$$\text{norm. abs. err.} = \frac{\sum_{i=1}^{N} |\hat{n}_i - n_i|}{\sum_{i=1}^{N} n_i}$$

The last three metrics, which measure entire groups of counter estimates, are applied only to the top N counter estimates.

Recall (see Section 2.1), we have 4 table types (src IP, dst IP, src ProtoPort, and dst ProtoPort), each of which has 3 counters (packets, bytes, and flows). We evaluate each of these 12 sets of estimates with the three error metrics above, giving 36 values. We also evaluate relative error for each of the 3 global counters. So, including memory and CPU, we measure 41 values for each system configuration. For brevity, when comparing different system configurations in this paper we will focus on the subset of values that are significantly different between those configurations. Tables of all of the values for all system configurations discussed in this paper are available in the technical report version of the paper [22].

Although we have tested the system with multiple data sets, for simplicity we have chosen a few representative sets for the results we present here:

- "OC48-A" - an hour trace from Aug 2002 of one direction of traffic on an OC48 link located at Metromedia Fiber Network (MFN) in San Jose.
- "OC48-B" - an hour trace of one direction on another OC48 link at MFN in San Jose.
- "Campus" - an hour trace of inbound and outbound traffic at a large university campus, in May 2003.
- "Backbone" - a 5 minute trace of one direction of traffic on an OC48 IP backbone link, with an average rate of approximately 400 Mbps (the first 5 minutes of "OC48-A").
- "DDoS" - 10 million artificially generated 44 byte packets spread over 5 minutes (about 12 Mbps) with fixed dst IP and dst ProtoPort and random src IP and src ProtoPort, simulating a random-source distributed denial of service attack on a single victim. The DDoS data can be mixed with any of the other datasets.

Table 1 presents summarized characteristics of these traces.

6.2 Comparing Sample and Hold Variants

To compare the two algorithms for identifying important entries, PSH and FSH, we run them separately. Since the sampling rates of both algorithms can be used to trade off memory for accuracy, we repeat the experiments with a number of sampling rates. To separate the errors introduced by entries being allocated too late from those due to flow counting algorithms, we use exact flow counting algorithms based on hash tables. Figure 3 illustrates this tradeoff for each algorithm on the IP tables with the backbone + DDoS dataset. Metrics for the byte counters are omitted for brevity; they are similar to those for packet counters, as expected. The accuracy of flow summaries is generally better than that of packet summaries for FSH and worse for PSH. Also as expected, FSH alone is worse than PSH alone; but we will use them together to cancel out each other's weaknesses. The normalized absolute errors in the dst IP graphs are much smaller than the corresponding RMS relative errors because the system counted the flows of the single DDoS victim very accurately and its flow count dwarfs that of other destination hosts.

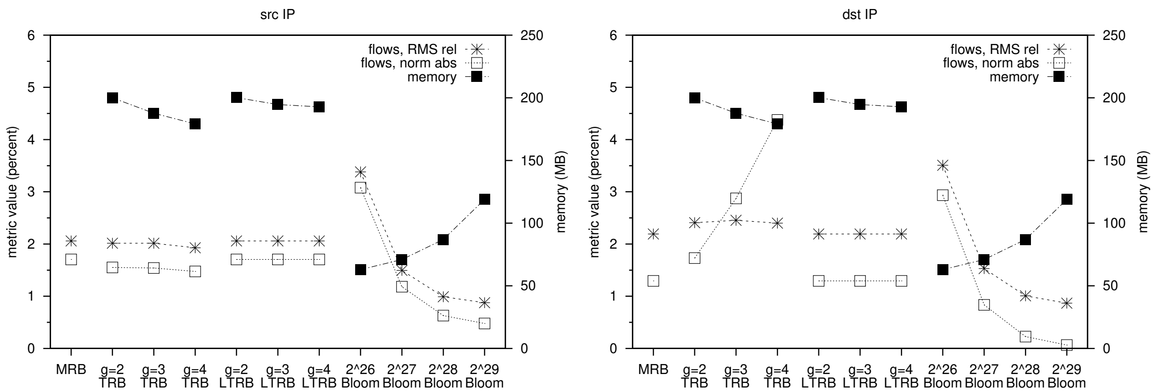

Figure 4: Comparison of flow counting algorithms: Multi-resolution bitmap (MRB), triggered bitmap (TRB), list-triggered bitmap (LTRB), and Bloom filter tuple set (Bloom), with fixed PSH rate of 1/128 and fixed FSH rate of 1/16. Memory use for MRB is off the graph at 1062 MB. Data: Backbone + DDoS.

6.3 Counting Algorithms and Configurations

To compare the various flow counting algorithms, we use fixed sampling rates of 1/128 for PSH and 1/16 for FSH; these appear reasonable from visual inspection of Figure 3. Once we have chosen a counting algorithm, we will switch to adaptive sampling rates.

Now we compare the different flow counting algorithms. Packet and byte counter estimates are plain integers, so their accuracy is determined entirely by the sampling rates; flows are the only values counted by our counting algorithms, so we analyze the accuracy and memory usage of only the flow reports here. We configure the bitmap algorithms so that their multiresolution bitmaps should give an error of about 3%. We expect the triggered and list-triggered bitmap algorithms to use less memory than the MRB algorithm, but because PSH and FSH will cause the omission of many small entries that would not trigger the creation of an MRB, this memory saving will be less dramatic than it would be without sampling. The trigger value g of the list-triggered bitmap algorithm will have no effect on the accuracy of large entries, so we try several reasonable values to see which uses the least memory. A list-triggered bitmap with $g = 2$ and 64 bit hash functions has 128 bits of trigger overhead. Since [15] gives no procedure for choosing triggered bitmap's direct bitmap size d or trigger value g, we start by matching the memory usage of the list-triggered bitmap just described by setting $d = 128$ and $g = 2$, then try several higher values of g which should save memory by avoiding allocation of some MRBs, but increase error. Finally, we try the Bloom filter tuple set algorithm with bit array sizes ranging from 2^{26} to 2^{29}. Thus the Bloom filters themselves use a fixed amount of memory between 8 MB and 64 MB, and the only variable affecting memory for a given configuration is the number of table entries.

First we consider the results shown in Figure 4 for the bitmap-based algorithms: multi-resolution bitmap (MRB), triggered bitmap (TRB), and list-triggered bitmap (LTRB). As expected, the memory usage of MRB is impractically large under attack situations, but the other two bitmap algorithms have much lower memory usage. TRB and LTRB perform roughly the same most of the time, but the error of TRB increases significantly under some conditions. On the other hand, LTRB's accuracy is very consistent and not susceptible to parameter choice; a bad choice would affect only memory, and that only slightly. These reasons, plus the fact that LTRB is slightly faster, make LTRB preferable over TRB.

For the Bloom filter tuple set algorithm, if the Bloom filter is too small, it becomes too densely filled under attack and gives inaccurate results. But with a reasonably sized Bloom filter, say 2^{28} bits, it gives quite accurate results even under extreme conditions, while using significantly less memory than any of the bitmap-based algorithms. Bloom filter tuple set appears more accurate than LTRB in these graphs, but they are both reasonable choices; indeed, under different traffic mixes or variations of this system with different tables, LTRB might do slightly better. However, the Bloom filter tuple set algorithm has several additional advantages. The table entries are smaller because they have a 64-bit counter whereas the bitmap algorithms have a 128-256 bit trigger and a many-hundred bit MRB, so its memory usage grows more slowly with increased traffic. The table entries are fixed size, unlike in TRB and LTRB which allocate large multiresolution bitmaps when triggered; this makes Bloom's memory usage closely related to the number of table entries, which is easier to control. The Bloom algorithm's significantly lower memory usage, less than half of other approaches with 2^{28} bits, makes it able to stand up better than the other algorithms under conditions even more extreme than our simulated DDoS attack. Additionally, the plot of errors by rank in Figure 5 is nearly monotonic for Bloom filter tuple set, meaning it is much better at correctly ranking the results. Finally, Bloom is significantly

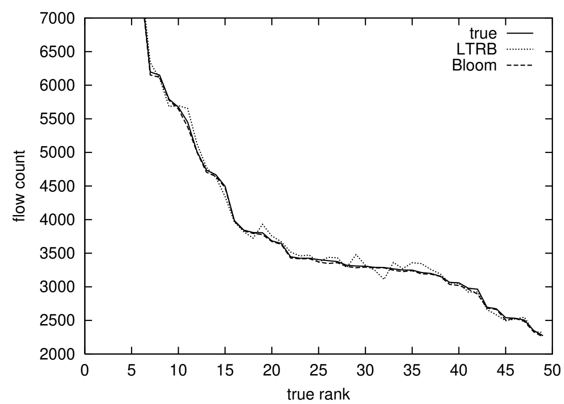

Figure 5: Flow estimates for src IP table by rank (zoomed in to show detail), using the same g=2 LTRB and 2^{28} Bloom runs as Figure 4. The Bloom curve is nearly monotonic, but starting at true rank 18, there are several significantly non-monotonic sections in the LTRB curve which would result in incorrect rankings.

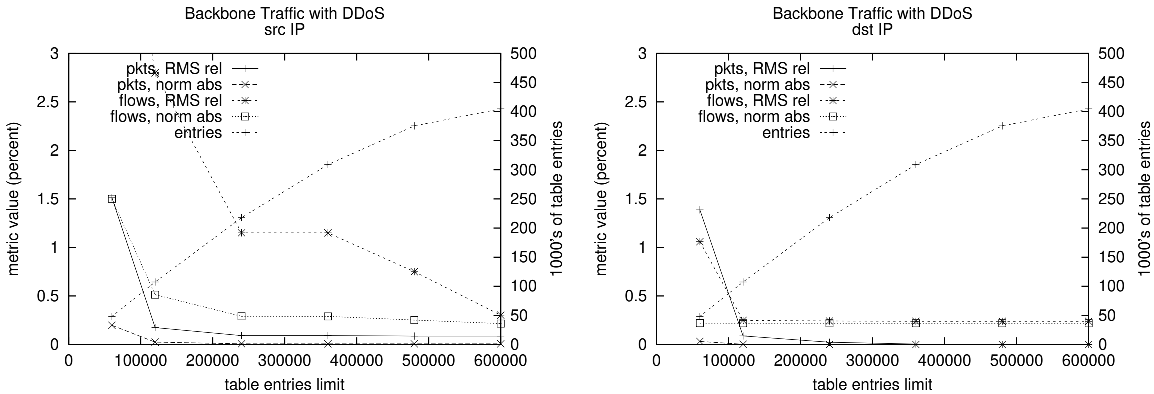

Figure 6: Effect of total table entries limit of adaptive algorithm on the accuracy of results and total number of entries allocated. Counting done with 2^{28}-bit Bloom filter with adaptive sample and hold rates.

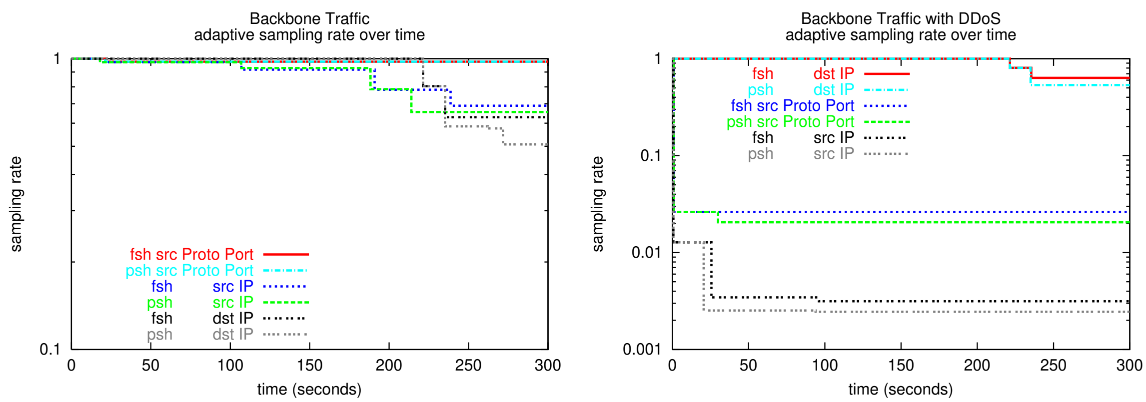

Figure 7: Adaptive sampling rate over the course of a measurement interval, with a 480k table entry limit and a 2^{28}-bit Bloom filter. Normal traffic causes very little reduction in sampling rates, but the system reacts quickly and accurately to tables filling due to a DDoS attack. (Note the y-axes have different scales.)

faster than any of the other algorithms; with a 2^{28} bit array it ran in 193 seconds, compared to 255 seconds for the fastest bitmap-based configuration (TRB with $g = 4$). For all these reasons, we chose Bloom filter tuple set as the counting algorithm for our system.

6.4 Adaptivity

Our system achieves robustness and resource isolation with respect to memory usage by adjusting the sampling rates of the PSH and FSH algorithms (see Appendix A for details) that are responsible for creating table entries (the only instance of allocating new memory). If that is not enough to also keep CPU usage under control, we adapt the pre-sampling rate (see the technical report [22] for details). In this section we will first see how different memory limits affect the accuracy of the results under a DDoS scenario. Next we fix the memory limit and measure how smoothly the sampling rates adapt and how they achieve isolation under DDoS traffic adverse to only some tables. Finally we look at how pre-sampling adaptation works under extreme attack scenarios.

In Figure 6, we see that error is kept low even during the simulated DDoS attack for a wide range of memory limits. At higher limits, the actual number of entries is somewhat lower than the limit, partially because our equal division of entries among the 8 table-samplers is not quite optimal, and partially because our memory usage prediction is not perfect. In future work, we plan to reallocate entry limits to each table for each interval based on how many entries they used in the previous interval. Above the 480,000 limit, the number of actual entries for normal backbone traffic (not shown) is nearly constant, because it was able to get all the entries it needed without reducing sampling rates below 100%. Even at very low limits, all metrics were rather good, with the obvious exception of the src IP flows in the DDoS test, since the DDoS creates an extremely large number of src IP entries. With limits of 240,000 or higher, even this hardest-hit metric is rather good, and at 480,000 and above, all error metrics are below 1%.

Figure 7 shows how the system has dynamically adapted the sample and hold rates over time to meet the memory constraint. With the backbone dataset, the number of entries allocated to most of the table-samplers was only slightly less than they wanted, so the system had to reduce their sampling rates only a little. On the other hand, DDoS traffic quickly created many entries in the src Proto Port and src IP tables, and the system reacted in less than a second (when each of the tables reached 1/4 capacity) by reducing their sampling rate. With just one more correction 20-30 seconds later, each of these samplers settled on a rate it was able to sustain for the remainder of the interval. The dst IP and dst Proto Ports samplers maintained the same high sampling rate in both situations, because the DDoS had little effect on them (the apparent difference in the dst IP curves is due to the change in the scale of the y-axis).

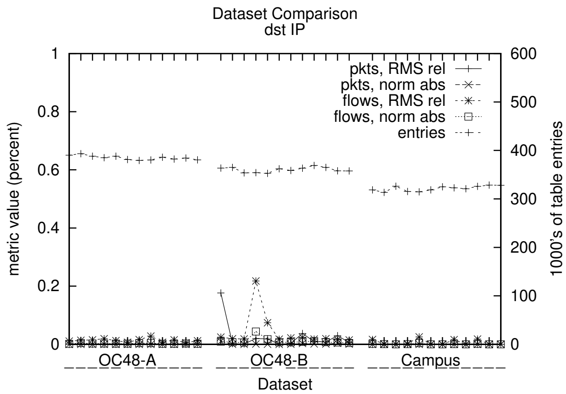

Figure 8: The system behaves consistently with different datasets. The (comparatively) large spikes in several intervals of OC48-B were due to an unusual network event, not a deviation in the system. Counting done with 2^{28}-bit Bloom filter and adaptive sample and hold rates with a 480k table entry limit.

We fix 480,000 entries as the limit for further runs of the system, since the error rates are very good here and this keeps memory usage for the tables within 32MB of RAM with our current C++ STL implementation. Combined with the 2^{28}-bit Bloom filter and miscellaneous overhead, this keeps total memory usage of the stack and heap below 64MB.

To handle much higher traffic rates, the system must resort to packet pre-sampling. Since there is nothing new about packet sampling, and our system can often avoid using it, detailed measurement of its effects are deferred to the technical report version of this paper [22].

6.5 Validation

Finally, we measured the system with different random seeds and different datasets to verify that its behavior is consistent.

We tested different seeds for the random number generator used to create the H3 hash functions, with a DDoS attack. Even the most stressed counter, src IP flows, maintained an RMS relative error of less than 2% with all seeds, and less than 1% with most seeds. The dst IP table is even more consistent. The results of the Proto Port tables are similar to those of the IP tables. More details of this experiment are available in the technical report version of this paper [22].

In Figure 8 we show the results of running the system with 12 different 5 minute samples of 3 different real traffic traces. Again the system demonstrates remarkable consistency; almost all of the error metrics are under 0.05%, even for the OC48-B dataset which contains approximately twice as much traffic as the Backbone dataset used up to now in this paper. The comparatively large (but still quite acceptable) RMS relative errors for flows in the fourth and fifth intervals of OC48-B are not a deviation in our system, but are in fact due to an unusual traffic event: whereas most of the intervals of OC48-B had around 2 million flows, intervals 4 and 5 had 6.7 million and 4.3 million.

7. CONCLUSIONS

In this paper we present a system that produces real-time summaries of Internet traffic. The main novelty of our system is that it achieves robustness to anomalous or malicious traffic mixes and isolation between the resource consumption of the modules computing different traffic summaries by adapting the parameters of algorithms. The types of summaries produced by our system are widely used by network administrators monitoring the workloads of their networks: the ports sending the most traffic (gives information about the applications in use); the IP addresses sending or receiving the most traffic (gives information about heavy users); the IP addresses with the most flows (reveals the victims of many denial of service attacks and computers performing aggressive network scans); etc.

We evaluate many algorithmic solutions to the problem of identifying and accurately measuring sources and destinations with many flows. We propose two novel solutions, “flow sample and hold” and “Bloom filter tuple set counting”, that present specific advantages over prior solutions. In particular, flow sample and hold improves accuracy for traffic mixes for which packet sample and hold alone would be inaccurate. Compared to the next best flow counting algorithm, Bloom filter tuple set is faster, uses approximately half the memory, and is generally more accurate.

The summaries produced by our measurement system are more concise and accurate than the measurement results of current systems. Anomalous network behavior such as denial of service attacks or worms that could push resource consumption beyond the limits of the hardware is handled by our system through graceful degradation of the accuracy of the summaries. Our system was able to maintain an error rate of less than 2% for all summaries on trace data from a lightly loaded OC-48 (2.5Gbps) combined with simulated denial of service attack, while using less than 1/20th the memory of a traditional system. With a 750 MHz processor, our system can handle up to OC-12 speeds without pre-sampling packet headers. If the traffic is larger, or malicious, pre-sampling protects the CPU, without drastically affecting the accuracy of summaries reporting byte counts and packet counts.

The evaluation of our system shows that it is feasible to build robust systems for computing accurate Internet traffic summaries in real time. In particular, by combining appropriate identification and flow counting algorithms with adaptive control, our system is able to compute multiple useful traffic summaries within affordable memory and CPU budgets at OC-48 speeds.

8. REFERENCES

[1] IPMON - packet trace analysis. `http://ipmon.sprintlabs.com/packstat/packetoverview.php`.

[2] PSAMP working group. `http://www.ietf.org/html.charters/psamp-charter.html`.

[3] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of the 6th International Workshop on Randomization and Approximation Techniques in Computer Science*, 2003.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Commun. ACM*, volume 13, pages 422–426, July 1970.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Journal of Computer and System Sciences*, volume 18, Apr. 1979.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? pages 436–447, June 1998.

[7] G. Cormode and S. Muthukrishnan. What’s hot and what’s not: Tracking most frequent items dynamically. In *In Proceedings of PODS*, June 2003.

[8] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk.

Gigascope: A stream database for network applications. In *p-sigmod*, June 2003.

[9] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Usenix Security*. Usenix, Aug. 2003. .

[10] N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *SIGCOMM Internet Measurement Workshop*, Nov. 2001.

[11] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *SIGCOMM*, pages 325–336, Aug. 2003.

[12] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, Sept. 2003.

[13] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *SIGCOMM*, Aug. 2004.

[14] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, Aug. 2002.

[15] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Internet Measurement Conference*, Oct. 2003.

[16] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases*, pages 307–317, Aug. 1998.

[17] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. In *SIGCOMM*, pages 257–270, Aug. 2000.

[18] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, Oct. 1985.

[19] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. pages 331–342, June 1998.

[20] Intel Corporation. E7505 Chipset. `http://www.intel.com/design/chipsets/e7505/`.

[21] Intel Corporation. E8870 Chipset. `http://www.intel.com/design/chipsets/e8870/`.

[22] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of Internet traffic: Technical report. `http://www.caida.org/outreach/papers/2005/tr-2005-01/`.

[23] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and k claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 (Passive and Active Measurement Workshop)*, Apr. 2001. .

[24] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *Proc. of IEEE INFOCOM*, Mar. 2004.

[25] D. Moore, K. Keys, R. Koga, E. Lagache, and k. claffy. CoralReef software suite as a tool for system and network administrators. In *Usenix LISA*, San Diego, CA, 4-7 Dec 2001. Usenix. .

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

[27] Sampled NetFlow. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s11/12s_sanf.htm`.

[28] ServerWorks, Inc. GC-LE Chipset. `http://www.serverworks.com/products/GCLE.html`.

[29] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, Feb. 2005.

[30] Waikato Applied Network Dynamics group. The DAG project. `http://dag.cs.waikato.ac.nz/`.

[31] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

APPENDIX

A. ADAPTING THE PSH SAMPLING RATE

To control memory usage, [14] adjusts the sampling rate from one measurement interval to the next based on actual memory usage. This approach is vulnerable to memory overflowing during individual measurement intervals when the traffic mix changes suddenly. When the memory overflows, no new entries can be created for the remainder of the measurement interval, and important sources of traffic can go unnoticed. We aim to build a more robust adaptation mechanism that achieves good results even within the intervals in which the traffic gets suddenly worse[2]. We achieve this goal by adjusting sampling rate *within* measurement intervals.

We first describe our algorithm for adapting the sampling rate within a measurement interval assuming we have a single table, say keyed on the source IP address, and a single algorithm, say PSH, creating the entries. Later we discuss how we generalize it to multiple parallel sample-and-hold algorithms and to multiple tables.

Assuming that our system has enough memory to handle every packet of "normal" traffic without exceeding its memory limit, we begin each interval with the sampling rate set to 1. This means that for every packet we create a new entry in the src IP table if that packet's src IP does not already exist in the table. Our first goal is to make sure that if the traffic mix changes, we stay within our available memory by decreasing the sampling rate. Furthermore, we want to achieve this goal with minimal loss of accuracy. Therefore we want to reduce the sampling rate no further than is necessary to ensure that we will not exceed our available memory.

One way of approaching this problem is to divide the measurement interval into multiple smaller subintervals, and based on the observed behavior in earlier subintervals adjust the sampling rates. This is not a robust solution because a sudden spike of malicious traffic could use all of the available memory before the current subinterval ends. We could defend against this problem by making the subintervals very small, but we are still vulnerable in the last few subintervals unless we put aside big chunks of the table as a safety buffer. Additionally, very small subintervals would be too sensitive to random short bursts in the traffic and could cause us to adapt the sampling rate erratically.

Instead, we address the adaptation of sampling rates by dividing the available memory into smaller budgets. When the number of allocated entries reaches the current budget, we look at the rate at which entries have been allocated and decide whether we need to adjust the sampling rate: we decrease the sampling rate if and only if we expect the memory to run out before the end of the measurement interval, based on the growth rate of the table since the last rate adjustment. Thus when the traffic mix suddenly changes and PSH starts allocating entries very quickly, the table will exhaust the budget quickly and the algorithm will promptly reduce the sam-

[2] [14] uses 5 second measurement intervals, not 5 minute intervals like we do, which makes the problem of having incomplete data for a couple of measurement intervals less grave.

PREDICT_FILL_TIME(*halftimes*[ ])
if *halftimes*$[2] >$ *halftimes*$[1]$
slowdown $=$ *halftimes*$[2] -$ *halftimes*$[1]$
else
slowdown $= 0$
endif
oneeighthfilltime $=$ *halftimes*$[2]$
timeleft $= 0$
for $i = 3$ to 8
oneeighthfilltime += *slowdown*
timeleft +=*oneeighthfilltime*
endfor
return *timeleft*

Figure 9: Estimating the time it takes to fill up the other six eighths of memory based on the times it took to fill up the first two eighths (i.e. the two halves of the budget).

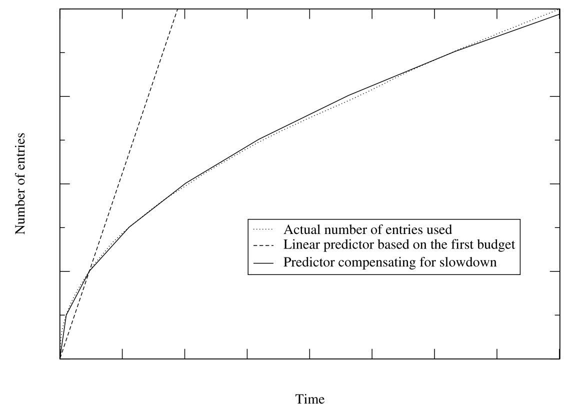

Figure 10: Using a linear prediction that assumes that the rate at which entries will be created is the same as the rate at which they were created during the current budget underestimates the time it takes to fill the memory. Accounting for the fact that it takes progressively longer to fill up the memory as we advance in time gives a much better prediction.

pling rate. In choosing the sizes of the budgets we need to balance two competing considerations: if they are too small the algorithm can overreact to small random spikes in the traffic, but if they are too large the algorithm will react too slowly. Furthermore, near the end of the interval we want to react more promptly because we have less memory left and unfriendly traffic can consume it faster. Our algorithm solves this problem by using budgets that are one quarter of the remaining available memory[3]. So the first budget is one quarter of the available memory, the next is one quarter of the remaining memory (three sixteenths of the total memory), and so on. Also, to avoid using very small budgets towards the end of the measurement interval, we perform the adaptation so that memory should run out slightly *after* the end of the measurement interval. When we adapt the sampling rate we pretend that the time the memory must last is 10% longer than it actually is, so that we will still have some memory left by the end of the actual memory interval. The size of the last budget will be no smaller than one quarter of this remaining memory which is the amount of memory we expect to fill up during the 10% "extension" to the measurement interval. The full adaptation algorithm is presented in the technical report version of the paper [22].

This adaptation algorithm needs to make a prediction of the rate at which table entries are going to fill up at the current sampling rate. This prediction is implemented by the "*predict_fill_time*()" function in Figure 9. While the prediction need not (and can not) be exact, the adaptation is more efficient if the prediction is close to the actual behavior: predicting that the memory will run out a lot sooner than it actually would will prompt the adaptation algorithm to decrease the sampling rate unnecessarily thus reducing the accuracy of the results, whereas predicting that the memory will run out much later than it actually would can consume the memory prematurely, forcing the algorithm to drastically reduce the sampling rate later on. Since we never increase the sampling rate within a measurement interval, we want to be especially careful that we do not severely underestimate the time it will take for the memory to fill up.

The simplest way of predicting when the memory will run out is to assume that the rate at which entries were allocated during the current budget period will continue. Figure 10 shows the number of entries created (without sampling) during a typical measurement interval. The figure clearly shows that linear prediction is very far from reality, because the rate at which entries are created slows down as the time progresses because there are fewer and fewer new source IP addresses in the traffic mix. However, with higher sampling rates, the memory usage curve is much closer to a straight line. We need a simple predictor that works for both cases. Our predictor achieves this by measuring the rate of the slowdown in the memory usage: we measure the time it takes to use up the first and second halves of the budget and store these times in *halftimes*$[1]$ and *halftimes*$[2]$. The difference between the two is the *slowdown*. We predict that the time it takes the algorithm to consume each eighth of memory is *slowdown* longer than the time to use the previous eighth. Therefore the third eighth should take *halftimes*$[2] +$ *slowdown*, the fourth should take *halftimes*$[2] + 2 *$ *slowdown*, and so on, and the total time to use up all six of the remaining eighths should be $6 *$ *halftimes*$[2] + 21 *$ *slowdown*. Figure 10 also plots this new prediction which is much closer to reality. Note that our prediction algorithm enforces that the slowdown is nonnegative (so it is never a "speedup"). If the second half of the budget is used up more quickly than the first half (due to an attack, for example), we use a slowdown of zero and thus base the prediction linearly on the rate at which only the second half of the budget was used.

Remember that our actual measurement system has four tables, not one, and two algorithms, PSH and FSH, operating on each table. We extend the algorithm presented here in a straightforward manner to this situation. The available entries are divided equally among the tables (but this can be overridden by the user through configuration), and furthermore the entries of each table are divided equally among the two algorithms. If both algorithms sample a packet that causes the creation of a new entry, they each get credit for half an entry. The rate adaptation for the eight samplers (two for each of the four tables) proceeds independently, thus achieving isolation between the measurement tasks.

[3]We also experimented with other fixed fractions such as one half or one eighth, but one quarter seemed to offer the best balance between responsiveness and stability.