An Adaptive System for Real-time Summaries of Internet Traffic

Cristian Estan[†], Ken Keys[*], David Moore[*,†] [*†‡]

Abstract

Good performance under excessive 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 system that computes multiple summaries of IP traffic in real time and achieves these design goals in a novel way: by automatically adapting parameters of the summarization algorithms. Anomalous network behavior, such as denial of service attacks or worms could push CPU or memory consumption beyond the limits of the hardware exactly when measurement is needed the most. 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 propose a new solution: "flow sample and hold". 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.

1 Introduction

The Internet service model grants great flexibility to the endnodes and imposes few restrictions on how they use the network. 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 weapon in each network administrator's arsenal 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

[*]E-mail: cestan@ucsd.edu, {kkeys,dmoore}@caida.org.

[†]$*$ - CAIDA, San Diego Supercomputer Center, University of California, San Diego. † - Computer Science and Engineering Department, University of California, San Diego.

[‡]Support for this work is provided by NSF Grant ANI-0137102, the Sensilla project sponsored by NIST Grant 60NANB1D0118, DARPA FTN Contract N66001-01-1-8933, and Caida members.

packet traces or packet header traces 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 [19] that aggregates all packets belonging to the same flow into a flow record. The current trend towards more 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 many flows have few packets, the 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 caused memory to be exhausted. NetFlow uses sampling at the router [20] 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 reduces the accuracy of the measurement results.

This paper addresses the problem of designing a robust traffic measurement system that handles unfriendly traffic mixes gracefully, by degrading the accuracy of the results it produces. 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: the specific summaries it needs to compute as well as more abstract goals such as robustness in the face of adverse traffic mixes and isolation between the resource consumption of the algorithms computing the various summaries. 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 new algorithm: 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. We conclude with Section 6.

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 Produced by the System

The traffic summaries we want our system to produce can be divided into two categories: global counters and top N reports. These reflect the traffic of the link monitored by the measurement system over fixed size measurement intervals. The 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.

The measurement system produces four types of top N reports keyed by specific packet header fields: source IP, destination IP, source port and protocol, and destination port and protocol. The system produces three "top N" reports for each key: with respect to the number of bytes, the number of packets, and the number of flows. The system produces a total of 12 top N reports with (up to) N items each. For example, we will have a report that lists the source IP addresses that sent the most packets, and how many packets each of those addresses sent.

We acknowledge that some network administrators may be interested in summaries different from the ones we compute. 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 IP addresses they connect to) or in-degree of destination addresses instead of flow counts. However, the methods we use could be readily applied to those measurements. Furthermore, with a software based architecture, 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 [21]. We identify the following four resources that can become 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 transferred across the network.

We exhaustively address the problem of building a system that is robust with respect to memory usage by using memory-efficient counting algorithms and accurate sampling techniques.

We present implementation techniques aimed at reducing the CPU usage, but the CPU of the system can still be overwhelmed by a large number of packets, as can the system bus. We can protect against this problem by performing some kind of sampling like sampled NetFlow [20] or packet selection [1] directly on the card. Since the card does not currently support this and we cannot change how it works, we use simulations to evaluate how adapting sampling can provide robustness with respect with CPU usage. While we present our solutions in the context of an architecture using a general purpose computer with a capture card, we expect many of our techniques to prove useful for measurement systems that use different technologies but have a subset of the same potential bottlenecks, for example devices built around a network processor.

Our system achieves robustness with respect to the potential bottlenecks by exploiting the fact that the summaries it produces can be approximate. However, we want to produce the best results attainable with the existing resources. When faced with atypical traffic that strains some resources of the system, it should continue providing summaries, though possibly at a lower accuracy. For example, a distributed denial of service (DDoS) attack with fake source addresses increases the number of active source addresses. This might exhaust the available memory by consuming too many entries in a table with per-source IP counters. The system could respond to this kind of traffic by refusing to create new entries in the source IP table, but this is not a good response because the source IP that sends the most traffic might start sending after this decision was made, and as a result the table will omit this interesting address altogether. Small errors in the counters and misordering source IPs with similar traffic are acceptable ways of degrading accuracy. Ignoring a source IP whose traffic is significantly above that of others included in the top is not.

Using the same processor to compute the various summaries and sharing memory between the algorithms computing them can result in cheaper and more efficient systems when compared to the alternative of complete isolation between the modules responsible for the summaries. However, we do not want a traffic mix that is adverse to one of the summaries 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 counters in the source IP entries, but the results in the destination IP reports (which would not be affected by the attack if implemented in isolation) should not be affected.

3 Related Work

NetFlow [19] 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 [14, 8]. To handle the volume and traffic diversity of high speed backbone links, NetFlow resorts to sampling [20]. The sampling rate is a configuration parameter set manually and seldom adjusted. Setting it too low results in very inaccurate measurement data; setting it too high can result in the measurement module using too much memory and processing power, especially when faced with increased or unusual traffic. 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. The Gigascope project [7] 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 identifying and measuring in a streaming fashion the addresses and ports with heavy traffic, and counting the number of flows. Identifying heavy hitters had been addressed in both database [16, 13, 6] and networking [11] context. Counting the number of distinct items has been addressed [15, 22, 2, 10] by the database community. The related problem of finding the sources/destinations with many flows and counting the flows they have can be easily solved by using hash tables that explicitly store all flow identifiers [17, 18], but the memory cost can be excessive. More efficient bitmap algorithms have been proposed [12]. Our system directly uses some of them while improving on others.

4 Algorithms for computing 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 IP addresses ports and flows we use multiresolution bitmaps[12]. Since these have low and constant

CPU and memory usage and well understood accuracy, for the rest of the paper we will focus on the top N reports. A simple way of producing a top N report say for source IP addresses is to keep a hash table with a counter for each source IP address in the traffic mix and just report the top ones at the end of the measurement interval. There are two problems with this approach: we cannot use simple counters in each entry to count flows and the table can get too large. Section 4.1 discusses the algorithms we use for counting flows and Section 4.2 those for identifying the important entries worth keeping in the tables.

4.1 Flow counting

Counting flows is not as easy as counting packets or bytes. The byte and packet counters of each table entry are updated on each packet that matches the entry. Flow counters need be updated only if the packet belongs to a new flow. Thus they must somehow distinguish between packets belonging to old flows and those that start new ones. The problem becomes even harder if we want to count separately the number of flows belonging to each of the source IP addresses we track.

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 table keyed by flow ID (or "tuple") 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 [17] [18] `crl_flow` and `t2_report`. It gives exact counts, but uses a large amount of memory to store the tuple table, and concentrates much of the processing cost at the end of the interval. Under extreme conditions, such as a worm attempting to spread or a distributed denial of service attack, the tuple table gets extremely large and this algorithm fails 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 [22] or direct bitmap [12], 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 [12] and probabilistic counting [15] 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 thousands of bytes these algorithms can give estimates with average errors of around 3% for up to hundreds of millions of flows.

4.1.2 Triggered Bitmaps

Instead of a global tuple table a simple flow counter in each entry of the target tables, we can add a multiresolution bitmap 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 [12] 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.

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 using 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 denial of service attacks with source addresses faked at random or worms aggressively probing random destinations, keeping even small amounts of memory for each unique source and destination IP address can consume too much memory for even a generously endowed workstation. Thus while we want to keep state for the important source and destination IPs, we cannot afford to keep state for all of them. We need algorithms to identify the addresses for which to keep entries.

4.2.1 Sample and hold

The sample and hold algorithm [11] can be used to identify and accurately measure the sources that send many packets and avoid keeping state for most of the sources that send little traffic. Packets are sampled at random, an entry is created for the source IP of each sampled packet (unless it already has one) and all packets belonging to that source are counted from there 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 off memory for accuracy: a high sampling probability gives accurate results, a lower one reduces memory usage but increases the errors in the counters because more packets go by uncounted before the entry is created.

The analysis of PSH [11] shows that it identifies with high probability the sources (and destinations) that send many packets. A simple line of thought seems to confirm that PSH could also be used to identify source IP addresses with many flows, 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. Say that we have a traffic mix consisting of peer to peer traffic moving 1.5 megabyte songs (consisting of 1000 maximum sized packets) from random IP addresses to random IP addresses along with a few port scanners scanning at a rate of 50 destinations per 5 minute interval. Of course we want to detect the port scanners, because those sources have the largest number of active flows. In 5 minutes there is room for approximately 60,000 1.5 megabyte files on an OC-48. Say we only have space for 50,000 entries in the source IP table. Thus if we use PSH, we can sample at most one packet in 1200 to ensure that the table doesn't fill up. At this packet sampling rate the probability of catching a port scanner is approximately 1/24. We are not catching the port scanners because the probability for 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 flows that have many packets will make it hard for PSH to catch early enough the sources that have many flows with few packets in each. While admittedly the traffic mix from our example above is not typical, since we want to build a system that is robust in the face of changes in the traffic mix, we need an algorithm that doesn't have such an unpalatable failure mode.

4.2.2 Flow sample and hold

Flow sample and hold (FSH) is similar to packet sample and hold (PSH), but it uses a sampling function that makes it more likely for sources with many flows to get an entry in the source table. We hash the flow identifiers of packets; for the packets whose identifiers fall into a certain subinterval of that hash space, we create an entry in the source IP table. Say the hash on the flow ID produces 32 bit hash values. We keep a variable f that controls the rate at which flows are sampled. If the hash value is below f, an entry is created. Setting f to 2^{32} amounts to FSH creating an entry for each flow. As we use lower values for f , fewer entries are created. The flow sampling probability of FSH is $f/2^{32}$. Notice that all packets belonging to a flow will have the same hash value, thus each flow has the same probability of triggering the creation of an entry for its source IP, irrespective how many packets it has. Furthermore, as the number of flows a source has increases the probability of it not getting an entry that tracks it decreases exponentially. Therefore sources with many flows will get entries early on and the flow counter in that entry will count all further flows initiated after the entry is created (and even the old ones if they send more packets). For the example above, let's say we use flow sampling with a sampling probability of 0.1. The probability that a flow scanner gets an entry is $(1 - 0.9^{50}) \approx 95.5\%$. At the same time the sources of peer to peer traffic that have one flow each will have a probability of 0.1 of receiving an entry. Thus the peer to peer traffic will add only around 60,000*0.1= 6,000 entries to the source IP table.

Can flow sample and hold replace packet sample and hold? Is it guaranteed to catch sources sending many packets (or bytes)? The answer is no. Imagine that we also have in the traffic mix from above someone only sending a 100 megabyte file through a single TCP connection. This is by far the largest sender, but if the single TCP connection doesn't hash onto the portion of the hash space that triggers addition to the source IP table (and this will happen with a probability of 90% using the parameters above), we will not create an entry and thus ignore this important source. The obvious answer for a system that aims to detect the sources that send many packet and also those that have many flows is to use both PSH and FSH to populate the tables with entries.

The question that arises naturally is whether we need yet another algorithm for detecting the sources that send many bytes of traffic. The reason why we need both PSH and FSH is that the ratio between the number of packets in two flows can be into the thousands. However, the ratio between the number of bytes in two packets does not exceed 40 in current traffic mixes as packets range in size from 40 to 1500 bytes. While using sampling rates that depend on packet sizes as in [11] does lead to more accurate results than relying on PSH (which samples packets with equal probability), since the ratio between packets 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, the difference is small. For simplicity and efficiency, we decided to use PSH to also identify the sources (and destinations) with many bytes of traffic.

5 System description

In this section we describe the actual measurement system we implemented. We first describe in Section 5.1 how an individual component that computes one top N report can adapt to the traffic mix and avoid exhausting the memory or the CPU. In Section 5.2 we describe the structure of the full system and show how sharing between components helps. 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 address the problem of how a component that only computes the report of the top N source IP addresses with respect to the number of packets sent can achieve robustness with respect to memory and CPU usage when faced with adverse traffic. The problems of computing the other 11 top N reports robustly are similar and we will point out the differences where they matter.

The basic idea is to keep a table with an entry for each source IP address and count the packets they send. This solution can lead to excessively large tables for adverse traffic mixes such as massive denial of service attack with source addresses faked at random. Using packet sample and hold to restrict the number of entries created can solve this problem but we need to determine the right sampling rate: if we sample too often we create too many entries, and if we sample too rarely we introduce unnecessary inaccuracy in the results. Our solution is to use an adaptive PSH sampling rate that we adjust based on how quickly the memory is filling up. We start each measurement interval with a PSH sampling probability of 1: we create an entry for each distinct source IP in the traffic stream. If entries are being allocated too quickly, we decrease the PSH probability to a value that we estimate will ensure that the table size stays within the memory budget until the end of the measurement interval. Appendix A gives the details of the method we use to adapt the PSH sampling probability. When using 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 the memory usage.

While adapting the PSH sampling probability allows us to control memory usage we still need to do a lookup in the table for each incoming packet. At OC-48 line speeds the CPU has 128 nanoseconds to process a 40 byte packet. Packet buffers can absorb bursts of packets, but the CPU cannot keep up with long streams of back to back short packets that can come for example from a massive distributed denial of service attack. We defend against this problem by adaptively performing a random pre-sampling of the packets on the capture card, before they reach the CPU. The details of the method we use to adapt the packet pre-sampling probability are work in progress. We compensate for the pre-sampling when we update the counters. For example if we pre-sample with probability 1 in 5, when a packet passes pre-sampling we update the counter of its table entry of by 5 instead of 1. The randomness of pre-sampling does introduce errors in the counters, but for sources with a large number of packets (i.e. the sources we are most interested in), the relative error is small.

Pre-sampling also reduces the number of entries in the table since many of the sources with few packets will have all their packets dropped, so we could control both CPU and memory usage by adjusting the packet pre-sampling probability, without using PSH. However, the analysis of the algorithms [11] shows that PSH gives results with much better accuracy than pre-sampling when they use the same sampling probability (and thus have the same memory consumption). Therefore, to maximize the accuracy of the results, we use PSH to control memory use, and only resort to pre-sampling if necessary to keep the CPU usage under control.

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 the flow reports. It has been shown that for any estimator that computes an estimate of the number of flows 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[5]. For example if we divide the number of flows we count in the pre-sampled packets by the pre-sampling probability, we get accurate flow counts if the flows contain a single packet, but we severely overestimate if all flows have many packets (and thus they all have at least one packet pass pre-sampling). If we do not adjust the flow count, we underestimate when the traffic mix has short flows. Using additional information such as TCP SYN flags in the pre-sampled packet headers can lead to more robust estimates [9]. These methods apply only to TCP and rely on the endhosts correctly setting the header 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 tables counting flows: we do not compensate for the pre-sampling. While this does result in underestimating the number of flows when pre-sampling is aggressive and the mix contains short flows, we at least know that our flow counts are a lower bound on the actual number of flows associated with the sources we track.

5.2 Structure of the system

Our system integrates the modules that compute the various summaries. Sharing between these modules often results in savings in memory and CPU usage which allow us to use more aggressive parameters thus achieving better accuracy. Note that for all

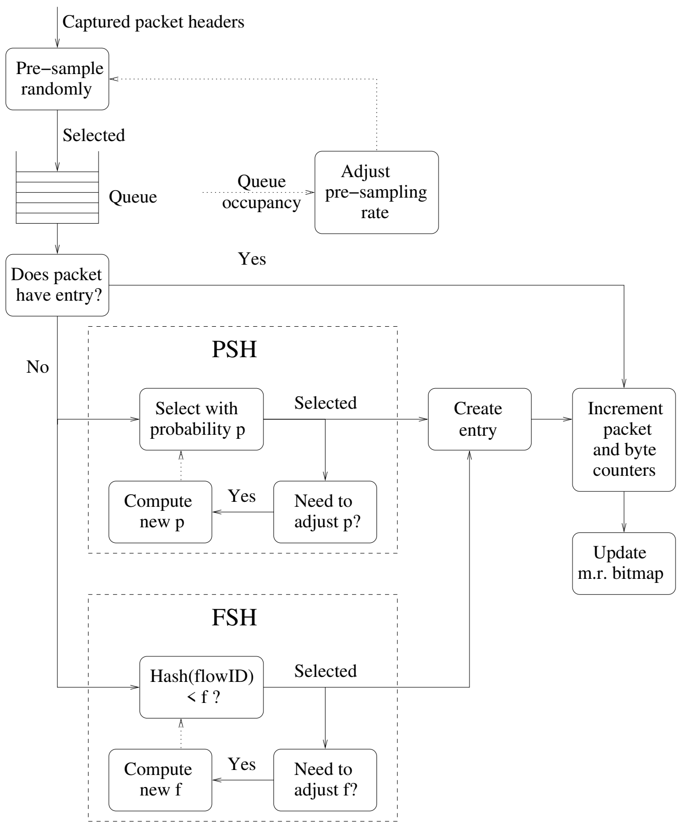

Figure 1: Tables keyed on various fields combine the algorithms for selecting entries (PSH and FSH). The sampling rates of PSH and FSH are adapted dynamically to ensure that the system never runs out of memory by allocating too many entries.

four types of top N report keys we produce three reports, by packets, bytes, and flows. 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 using combined tables is that we only have to perform four table lookups per packet instead of twelve, which amounts to significant savings in CPU usage. Figure 1 shows how a combined table operates.

Global counters are implemented with multiresolution bitmaps. We use an efficient implementation of the H3 hash function family[4], 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.

5.3 Isolation

Isolating the resource consumption of various components of the system allows us to ensure consistent accuracy. Memory is dynamically allocated only by the four tables used to compute the top N reports. Isolating the memory consumption of the four tables is easy: we divide among them the number of entries we can allocate. Thus while a denial of service attack with faked source IP addresses will strain the source IP table and the decrease in PSH and FSH sampling probabilities will reduce the accuracy of the source IP top N reports, the destination IP table will be unaffected and the destination IP reports will not loose accuracy. Our current system divides the memory equally among the tables, but this can be overridden through configuration. There is a simple improvement over the strategy of dividing memory between tables equally, 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 automatically redistribute the surplus among the others.

We have two algorithms, PSH and FSH, adding entries to each table. We also need to protect these two algorithms from each other. We achieve this by dividing the memory budget of each table equally among the two algorithms and adjusting the sampling probabilities of PSH and FSH independently (see Figure 1). If both algorithms sample a packet that causes the creation of a new entry, they each get credit 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. First of all 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 because of some traffic pattern. Therefore all modules need to reduce their CPU usage at the same time: when there are too many packet headers to process. The second reason to use a common packet pre-sampling stage is that for it to effectively protect the CPU and the bus it must be implemented on the capture card[1].

6 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.

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

We evaluate many algorithmic solutions to the problem of identifying and accurately measuring sources and destinations with many flows. The novel solutions we propose, "flow sample and hold", presents 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.

The summaries produced by our system measurement 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.

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.

References

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

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

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

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. In *Journal of Computer and System Sciences*, volume 18, April 1979.

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

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

[7] Chuck Cranor, Theodore Johnson, Oliver Spatschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *p-sigmod*, June 2003.

[8] Nick Duffield, Carsten Lund, and Mikkel Thorup. Charging from sampled network usage. In *SIGCOMM Internet Measurement Workshop*, November 2001.

[9] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *SIGCOMM*, pages 325–336, August 2003.

[10] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *ESA*, September 2003.

[11] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, August 2002.

[12] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Internet Measurement Conference*, October 2003.

[13] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases*, pages 307–317, August 1998.

[14] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational IP networks: Methodology and experience. In *SIGCOMM*, pages 257–270, August 2000.

[15] 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.

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

[17] Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 — A workshop on Passive and Active Measurements*. CAIDA, RIPE NCC, April 2001. http://www.caida.org/tools/measurement/coralreef/.

[18] 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. `http://www.caida.org/outreach/papers/2001/CoralApps/`.

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

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

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

[22] 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.

A Adapting the sampling rate for PSH

To control memory usage, [11] 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 are created for the rest 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 for the intervals where 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 that 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 it can be generalized 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 (for example we could trivially ensure our first goal by setting the sampling rate to 0, but then our table would stay empty and provide no measurement results). 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 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 sampling 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 inter-

[2][11] uses 5 second measurement intervals, not 5 minute intervals like we do and this makes the problem of having incomplete data for a couple of measurement intervals less grave.

```
INIT_PSH
  $interval\_end = now + interval\_size$
  $abs\_fill\_time = now + 1.1 * interval\_size$
  $samplingrate = 1$
  $remaining\_entries = available\_entries$
  $budget = remaining\_entries/4$
  $halfbudget = remaining\_entries/8$
  $used\_entries = 0$
  $budget\_start\_time = now$
```

Figure 2: Initialization of per-table PSH state at the beginning of each interval.

```
DO_PSH($packet$)
  if $random(0, 1) \leq samplingrate$
    $CreateEntry(packet.srcIP)$
    $used\_entries + +$
    if $used\_entries < halfbudget$
      return
    else if $used\_entries = halfbudget$
      $halfbudget\_time = now - budget\_start\_time$
      return
    else if $used\_entries < budget$
      return
    endif
    $halftimes[1] = halfbudget\_time - budget\_start\_time$
    $halftimes[2] = now - halfbudget\_time$
    $remaining\_entries- = used\_entries$
    $est\_fill\_time = predict\_fill\_time(halftimes)$
    $fill\_time = abs\_fill\_time - now$
    if $est\_fill\_time < fill\_time$
      $samplingrate* = est\_fill\_time/fill\_time$
    endif
    $budget = remaining\_entries/4$
    $halfbudget = remaining\_entries/8$
    $used\_entries = 0$
    $budget\_start\_time = now$
    $time\_remaining = interval\_end - now$
  endif
```

Figure 3: Algorithm for packet sample and hold on the src IP table with dynamically adapted sampling rate, applied to each packet whose src IP does not already exist in the src IP table. The algorithm works similarly on the other tables.

```
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 4: 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).

val 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 has to be enough for 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. Our final algorithm for PSH with adapting sampling rate is initialized at the beginning of each interval as shown in Figure 2, and then the code in Figure 3 is applied to each packet that does not already have an entry in the table. The purpose of $halfbudget$ and $halftimes[\,]$ will be explained later.

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 4. 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

[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.

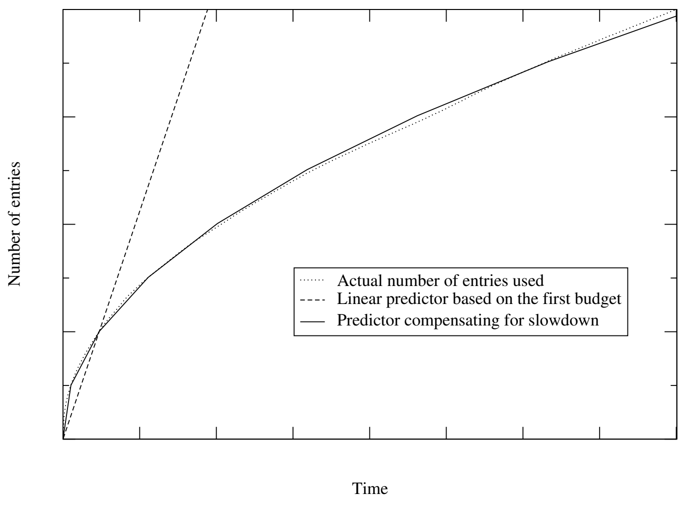

Figure 5: Using a linear prediction that assumes that the rate at which entries will be created is the same as the rate at which they were created during the current budget underestimates the time it takes to fill the memory. Accounting for the fact that it takes progressively longer to fill up the memory as we advance in time gives a much better prediction.

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 5 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 the previously mysterious $halftimes[1]$ 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 5 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 algorithms 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. There is a simple improvement over the strategy of dividing memory between tables equally, which we have not yet implemented: If some of the tables do not use up their allocated memory (e.g. the port tables), in the next interval we can automatically redistribute the surplus among the others.