Carousel: Scalable Logging for Intrusion Prevention Systems

Vinh The Lam[†], *Michael Mitzenmacher*[⋆], *George Varghese*[†]

[†]*University of California, San Diego* — [⋆] *Harvard University*

{*vtlam,varghese*}*@cs.ucsd.edu* — *michaelm@eecs.harvard.edu*

Abstract

We address the problem of collecting unique items in a large stream of information in the context of Intrusion Prevention Systems (IPSs). IPSs detect attacks at gigabit speeds and must log infected source IP addresses for remediation or forensics. An attack with millions of infected sources can result in hundreds of millions of log records when counting duplicates. If logging speeds are much slower than packet arrival rates and memory in the IPS is limited, *scalable logging* is a technical challenge. After showing that naïve approaches will not suffice, we solve the problem with a new algorithm we call Carousel. Carousel randomly partitions the set of sources into groups that can be logged without duplicates, and then cycles through the set of possible groups. We prove that Carousel collects almost all infected sources with high probability in close to optimal time as long as infected sources keep transmitting. We describe details of a Snort implementation and a hardware design. Simulations with worm propagation models show up to a factor of 10 improvement in collection times for practical scenarios. Our technique applies to *any* logging problem with non-cooperative sources as long as the information to be logged appears repeatedly.

1 Introduction

With a variety of networking devices reporting events at increasingly higher speeds, how can a network manager obtain a coherent and succinct view of this deluge of data? The classical approach uses a *sample* of traffic to make behavioral inferences. However, in many contexts the goal is *complete or near-complete collection* of information — MAC addresses on a LAN, infected computers, or members of a botnet. While our paper presents a solution to this abstract logging problem, we ground and motivate our approach in the context of Intrusion Prevention Systems.

Originally, Intrusion Detection Systems (IDSs) implemented in software worked at low speeds, but modern Intrusion Prevention Systems (IPSs) such as the Tipping Point Core Controller and the Juniper IDP 8200 [5] are implemented in hardware at 10 Gbps and are standard in many organizations. IPSs have also moved from being located only at the periphery of the organizational network to being placed throughout the organization. This allows IPSs to defend against internal attacks and provides finer granularity containment of infections. Widespread, cost-effective deployment of IPSs, however, requires using streamlined hardware, especially if the hardware is to be integrated into routers (as done by Cisco and Juniper) to further reduce packaging costs. By streamlined hardware, we mean ideally a single chip implementation (or a single board with few chips) and small amounts of high-speed memory (less than 10 Mbit).

Figure 1 depicts a logical model of an IPS for the purposes of this paper. A bad packet arrives carrying some key. Typically the key is simply the source address, but other fields such as the destination address may also be used. For the rest of the paper we assume the key is the IP source address. (We assume the source information is not forged. Any attack that requires the victim to reply cannot use a forged source address.) The packet is coalesced with other packets for the same flow if it is a TCP packet, normalized [16] to guard against evasions, and then checked for whether the packet is indicative of an attack. The most common check is *signature-based* (e.g., Snort [13]) which determines whether the packet content matches a regular expression in a database of known attacks. However, the check could also be *behavior-based*. For example, a denial of service attack to a destination may be detected by some state accumulated across a set of past packets.

In either case, the bad packet is typically dropped, but the IPS is required to *log* the relevant information on disk at a remote management console for later analysis and reporting. The information sent is typically the key K plus a report indicating the detected attack. Earlier work has shown techniques for high speed implementations of reassembly [4],

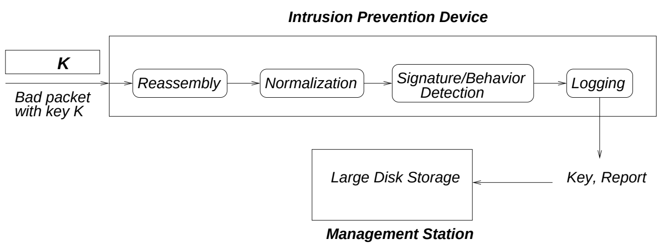


Figure 1: IPS logical model including a logging component that is often implemented naïvely

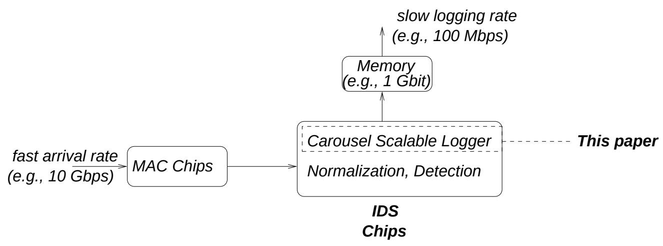


Figure 2: IPS hardware model in which we propose adding a scalable logger facility called Carousel. Carousel focuses on a small random subset of the set of keys at one time, thereby matching the available logging speed.

normalization [15, 16], and fast regular expression matching (e.g., [12]). However, to the best of our knowledge, there is no prior work in scalable logging for IPS systems or networking.

To see why logging may be a bottleneck, consider Figure 2, which depicts a physical model of a streamlined hardware IPS implementation, either stand-alone or packaged in a router line card. Packets arrive at high speed (say 10 Gbps) and are passed from a MAC chip to one or more IDS chips that implement detection by for example signature matching. A standard logging facility, such as in Snort, logs a report each time the source sends a packet that matches an attack signature and writes it to a memory buffer, from which it is written out later either to locally attached disk in software implementations or to a remote disk at a management station in hardware implementations. A problem arises because the logging speed is often much slower than the bandwidth of the network link. Logging speeds less than 100 Mbps are not uncommon, especially in 10 Gbps IDS line cards attached to routers. Logging speeds are limited by physical considerations such as control processor speeds and disk bandwidths. While logging speeds can theoretically be increased by striping across multiple disks or using a network service, the increased costs may not be justified in practice.

In hardware implementations where the memory buffer is necessarily small for cost considerations, the memory can fill during a large attack and newly arriving logged records may be dropped. A typical current configuration might include only 20 Mbits of on-chip high speed SRAM of which the normalizer itself can take 18 Mbits [16]. Thus, we assume that the logger may be allocated only a small amount of high speed memory, say 1 Mbit. Note that the memory buffer may include duplicate records already in the buffer or previously sent to the remote device.

Under a standard naïve implementation, unless the logging rate matches the arrival rate of packets, there is no guarantee that all infected sources will be logged. It is easy to construct worst-case timing patterns where some set of sources A are never logged because another set of sources B always reaches the IDS before sources in the set A and fills the memory. Even in a random arrival model, intuitively as more and more sources are logged, it gets less and less probable that a new unique source will be logged. In Section 3 we show that, even with a fairly optimistic random model, a standard analysis based on the coupon collector's problem (e.g., [8]) shows that the expected time to collect all N sources is a *multiplicative* factor of $\ln N$ worse than the optimal time. For example, when N is in the millions, which is not unusual for a large worm, the expected time to collect all sources can be 15 times larger than optimal. We also show similar poor behavior of the naïve implementation, both through analysis and simulation, in more complex settings.

The main contribution of this paper, as shown in Figure 2, is a scalable logger module that interposes between the detection logic and the memory buffer. We refer to this module and the underlying algorithm as *Carousel*, for reasons that will become apparent. Our logger is scalable in that it can collect almost all N sources with high probability with very small memory buffers in close to optimal time, where here the optimal time is N/b with b being the logging speed. Further, Carousel is simple to implement in hardware even at very high speeds, adding only a few operations to the main processing path. We have implemented Carousel in software both in Snort as well as in simulation in order to evaluate its performance.

While we focus on the scalable logging problem for IPSs in this paper, we emphasize that the problem is a general one that can arise in a number of measurement settings. For example, suppose a network monitor placed in the core of an organizational network wishes to log all the IP sources that are using TCP Selective Acknowledgment option (SACK). In general, our mechanism applies to any monitoring setting where a source is identified by a predicate on a packet (e.g., the packet contains the SACK_PERMITTED option, or the packet matches the Slammer signature), memory is limited, and sources do not cooperate with the logging process. It does, however, require sources to keep transmitting packets with the predicate in order to be logged. Thus Carousel does

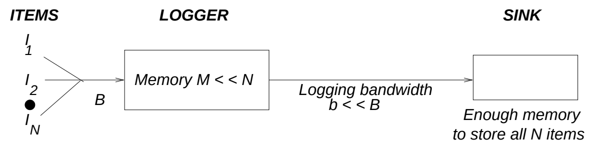


Figure 3: Abstract logging model: N keys to be logged enter the logging device repeatedly at a speed B that is much greater then the logging speed b and in a potentially adversarial timing pattern. At the same time, the amount of memory M is much less than the N, number of distinct keys to be logged. Source cooperation is *not* assumed.

not guarantee the logging of one-time events.

The rest of the paper is organized as follows. In Section 2 we describe a simple abstract model of the scalable logging problem that applies to many settings. In Section 3 we describe a simple analytical model that shows that even with an optimistic random model of packet arrivals, naïve logging can incur a multiplicative penalty of $\ln N$ in collection times. Indeed, we show this is the case even if naïve logging is enhanced with a Bloom filter in the straightforward way. In Section 4 we describe our new scalable logging algorithm Carousel, and in Section 5 we describe our Snort implementation. We evaluate Carousel using a simulator in Section 6 and using a Snort implementation in Section 7. Our evaluation tests both the setting of our basic analytical model, which assumes that all sources are sending at time 0, and a more realistic logistic worm propagation model, in which sources are infected gradually. Section 8 describes related work while Section 9 concludes the paper.

2 Model

The model shown in Figure 3 abstracts the scalable logging problem. First, there are N distinct keys that arrive repeatedly and with arbitrary timing frequency at a cumulative speed of B keys per second at the logger. There are *two* resources that are in scarce supply at the logger. First, there is a limited logging speed b (keys per second) that is much smaller than the bandwidth B at which keys arrive. Even this might not be problematic if the logger had a memory M large enough to hold all the distinct keys N that needed to be logged (using methods we discuss below, such as Bloom filters [1, 3], to handle duplicates), but in our setting of large infections and hardware with limited memory, we must also assume that $N >> M$.

Eliminating all duplicates before transmitting to the sink is *not* a goal of a scalable logger. We assume that the sink has a hash table large enough to store all N unique sources (by contrast to the logger) and eliminate duplicates.

Instead, the ultimate goal of the scalable logger is *near-complete collection*: the logging of all N sources. We now adopt some of the terminology of competitive analysis [2] to describe the performance of practical logger systems. The best possible logging time $T_{optimal}$ for an omniscient algorithm is clearly N/b. We compare our algorithms against this omniscient algorithm as follows.

Definition 2.1 *We say that a logging algorithm is (ϵ, c)-scalable if the time to collect at least $(1-\epsilon)N$ of the sources is at most $cT_{optimal}$. In the case of a randomized algorithm, we say that an algorithm is (ϵ, c)-scalable if in time $cT_{optimal}$ the expected number of sources collected is at least $(1-\epsilon)N$.*

Note that in the case $\epsilon = 0$ all sources are collected. While obviously collecting all sources is a desirable feature, some relaxation of this requirement can naturally lead to much simpler algorithms.

These definitions have some room for play. We could instead call a randomized algorithm (ϵ, c)-scalable if the expected time to collect at least $(1-\epsilon)N$ is at most $cT_{optimal}$, and we may be concerned only with asymptotic algorithmic performance as either or both of N/M and B/b grow large. As our focus here is on practically efficient algorithms rather than subtle differences in the definitions we avoid such concerns where the meaning is clear.

The main goal of this paper is to provide an effective and practical (ϵ, c)-scalable randomized algorithm. To emphasize the value of this result, we first show that simple naïve approaches are not (ϵ, c)-scalable for any constants $\epsilon, c > 0$. Our positive results will require the following additional assumption for our model:

Persistent Source Assumption: We assume that any distinct key X to be logged will keep arriving at the logger.

For sources infected by worms this assumption is often reasonable until the source is "disinfected" because the source continues to attempt to infect other computers. The time for remediation (days) is also larger than the period in which the attack reaches its maximum intensity (hours). Further, if a source is no longer infected, then perhaps it matters less that the source is not logged. In fact, we conjecture that no algorithm can solve the scalable logging problem without the Persistent Source assumption.

The abstract logger model is a general one and applies to other settings. In the introduction, we mentioned one other possibility, logging sources using SACK. As another example, imagine a monitor that wishes to log all the sources in a network. The monitor issues a broadcast request to all sources asking them to send a reply with their ID. Such messages do exist, for example the SYSID message in 802.1. Unfortunately, if all sources reply at the same time, some set of sources can consistently be lost.

Of course, if the sources could randomize their replies, then better guarantees can be made. The problem can be viewed as one of congestion control: matching the speed of arrival of logged keys to the logging speed. Congestion control can be solved by standard methods like TCP slow start or Ethernet backoff *if* sources can be assumed to cooperate. However, in a security setting we cannot assume that sources will cooperate, and other approaches, such as the one we provide, are needed.

3 Analysis of a Naïve Logger

3.1 The Naïve Logger Alone

Before we describe our scalable logger and Snort implementation, we present a straw man naïve logger, and a theoretical analysis of the expected and worst-case times. The theoretical analysis makes some simplifications that only benefit the naïve logger, but still its performance is poor. The naïve logger motivates our approach.

We start with a model of the naïve logger shown in Figure 4. We assume that the naïve logger only has a memory buffer in the form of a queue. Keys, which again are usually source addresses, arrive at a rate of B per second. When the naïve logger receives a key, it is placed at the tail of the queue. If the queue is full, the key is dropped. The size of the queue is M. Periodically, at a smaller rate of b keys per second, the naïve logger sends the key (and any associated report) at the head of the queue to a disk log. Let L_D denote the set of keys logged to disk, and L_M the set of keys that are in the memory.

The naïve logger works very poorly in an adversarial setting. In an adversarial model, after the queue is full of M keys, and when an empty slot opens up at the tail, the adversary picks a duplicate key that is part of the M keys already logged. When the queue is full, the adversary cycles through the remaining unique sources to pick them to arrive and be dropped, thus fulfilling the persistent source assumption in which every source must arrive periodically. It is then easy to see the following result.

Theorem 3.1 Worst-case time for naïve logger: *The worst-case time to collect all N keys is infinity. In fact, the worst-case time to collect more than M keys is infinite.*

We believe the adversarial models can occur in real situations especially in a security setting. Sources can be synchronized by design or accident so that certain sources always transmit at certain times when the logger buffers are full. While we believe that resilience to adversarial models is one of the strengths of Carousel, we will show that even in the most optimistic random models, Carousel significantly outperforms a naïve logger.

The simplest random model for key arrival is one in which the next key to arrive is randomly chosen from the N possible keys, and we can find the expected collection time of the naïve logger in this setting.

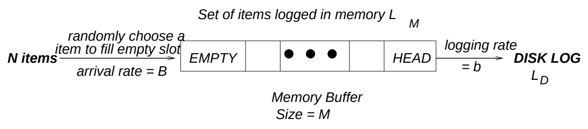


Figure 4: Model of naïve logging using an optimistic random model. When space opens up in the memory log, a source is picked uniformly and randomly from the set of all possible N sources. Unfortunately, that source may already be in the memory log (L_M) or in the disk log (L_D). Thus as more sources are logged it gets increasing less probable that a new unique source will be logged, leading to a logarithmic increase in collection time over optimal

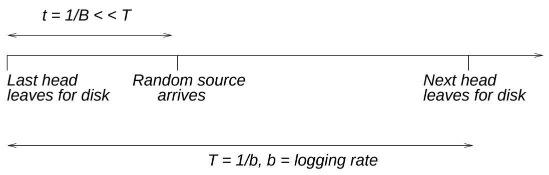

t = 1/B < < T
Last head leaves for disk
Random source arrives
Next head leaves for disk
T = 1/b, b = logging rate

Figure 5: Portion of timeline for random model shown in Figure 4. We divide time into cycles of time T where T is the time to send one piece of logged information at the logging rate b. The time for a new randomly chosen source to first arrive is much smaller $t = 1/B$, where B is the faster packet arrival rate.

Let us assume that $M < B/b$, so that initially the queue fills entirely before the first departure. (The analysis is easily modified if this is not the case.) Figure 5 is a timeline which shows that the dynamics of the system evolve in cycles of length T seconds, where $T = 1/b$. Every T seconds the current head of the memory queue leaves for the disk log, and within the smaller time $t = 1/B$, a new randomly selected key arrives to the tail of the queue. In other words, the queue will always be full except when a key leaves from the head, leaving a single empty slot at the tail as shown in Figure 4. The very next key to be selected will then be chosen to fill that empty slot as shown in Figure 5.

The analysis of this naïve setting now follows from a standard analysis of the coupon collector's problem [8]. Let $L = L_M \cup L_D$ denote the set of unique keys logged in either memory or disk. Let T_i denote the time for L to grow from size $i-1$ to i (in other words, the time for the i-th new key to be logged). If we optimistically assume that the first M keys that arrive are distinct, we have $T_i = T$ for $1 \leq i \leq M$, as the queue initially fills. Subsequently, since the newly arriving key is chosen randomly from the set of N keys, it will get increasingly probable (as i gets larger) that the chosen key already belongs to the logged set L.

The probability that a new key will not be a duplicate of $i-1$ previously logged keys is is $P_i = (N-i+1)/N$. If a key is a duplicate the naïve logger simply wastes a cycle of time T. (Technically, it might be $T - t$ where $t = 1/B$, but this distinction is not meaningful and we ignore it.) The expected number of cycles before the i-th key is not a duplicate is the reciprocal of the probability or $1/P_i$. Hence for $i > M, i \leq N$ the expected value of T_i is

$$\mathbf{E}(T_i) = \frac{N}{b(N-i+1)}.$$

Using the linearity of expectation, the collection time for the last $N - M$ keys is

$$\sum_{i=M+1}^{N} \frac{N}{b(N-i+1)} = \frac{N}{b} \sum_{j=1}^{N-M} \frac{1}{j} = \frac{N}{b} \left(\ln(N-M) + O(1)\right),$$

using the well-known result for the sum of the harmonic series. Hence if we let $T_{collect}^{naive}$ be the time to collect all N keys for the naïve collector then $T_{collect}^{naive} > \frac{N}{b}\ln(N-M)$, and so the naïve logger is a multiplicative factor of $\ln(N - M)$ worse than the optimal algorithm.

It might be objected that it is not clear that N/b is in fact the optimal time in this random model, and that this $\ln N$ factor is due entirely to the embedded coupon collector's problem arising from the random model. For example, if $B = b = 1$, then you cannot collect the N keys in time N, since they will not all appear until after approximately $N \ln N$ keys have passed [8]. However, as long as $B/b > \ln N$ (and $M > 1$), for any $\gamma > 0$, with high probability an omniscient algorithm will be able to collect all keys after at most $(1+\gamma)NB/b$ keys have passed in this random model, so the optimal collection time can be made arbitrarily close to N/b. Hence, this algorithm is indeed not truly scalable in the sense we desire, namely in a comparison with the optimal omniscient algorithm.

Even if we seek only to obtain $(1-\epsilon)N$ keys, by the same argument we have the collection time is

$$\frac{N}{b}\left(\ln((1-\epsilon)N - M) + O(1)\right).$$

Hence when $M = o(N)$, the logger is still not (ϵ, c)-scalable for any constants ϵ and c. We can summarize the result as follows:

Theorem 3.2 Expected time for naïve logger: *The expected time to collect $(1-\epsilon)N$ keys is at least a multiplicative factor of $\ln((1-\epsilon)N - M)$ worse than the optimal time for sufficiently large N, M, and ratios B/b.*

As stated in the introduction, for large worm outbreaks, the naïve logger can be prohibitively slow. For example, as $\ln 1,000,000$ is almost 14, if the optimal time to log 1 million sources is 1 hour, the naïve logger will take almost 14 hours.

The results for the random model can be extended to situations that naturally occur in practice and appear somewhere between the random model and an adversarial model. For example, suppose that we have two sets of sources, of sizes N_1 and N_2, but the first source sends at a speed that is j times the second. This captures, at a high level, the issue that sources may be sending at different rates. We assume each source individually behaves according to the random model. Let T_1 be the expected time to collect all the keys in the fast set, and T_2 the expected time for the slow set. Then clear the expected time to collect all sources is at least $\max(T_1, T_2)$, and indeed this lower bound will be quite tight when T_1 and T_2 are not close. As an example, suppose $N_1 = N_2 = N/2$, and $j > 1$. Then T_2 is approximately

$$\frac{N(j+1)}{2b} \ln\left(\frac{N}{2} - \frac{M}{j+1}\right).$$

The time to collect in this case is dominated by the slow sources, and is still a logarithmic factor from optimal.

3.2 The Naïve Logger with a Bloom Filter

A possible objection is that our naïve logger is far too naïve. It may be apparent to many readers that additional data structures, such as a Bloom filter, could be used to prevent logging duplicate sources and improve performance. This is true, and we shall use such measures in our scalable approaches. However, we point out that as the Bloom filter of limited size, it cannot by itself prevent the problems of the naïve logger, as we now explain.

To frame the discussion, consider 1 million infected sources that keep sending to an IPS. The solution to the problem may appear simple. First, since all the sources may arrive at a very fast rate of B before even a few are logged, the scheme must have a memory buffer that can hold keys waiting to be logged. Second, we need a method of avoiding sending duplicates to the logger, specifically one that takes small space, in order to make efficient use of the small speed of the logger.

To avoid sending duplicates, one naturally would think of a solution based on Bloom filters or hashed fingerprints. (We assume familiarity with Bloom filters, a simple small-space randomized data structure for answering queries of the form "Is this an item in set X" for a given set X. See [3] for details.) For example, we could employ a Bloom filter as follows. For concreteness, assume that a source address is 32 bits, the report associated with a source is 68 bits, and that we use a Bloom filter [1] of 10 bits per source.[1] Thus we need a total of 100 bits of memory for each source waiting to be logged, and 10 bits for each source that has been logged. (Instead of a Bloom filter, we could keep a table of hash-based fingerprints of the sources, with different tradeoffs but similar results, as we discuss in Section 4.2.2.)

[1]This is optimistic because many algorithms would require not just a Bloom filter but instead a counting Bloom filter [7] to support deletions, which would require more than 10 bits per entry.

Unfortunately, the memory buffer and Bloom filter have to operate at Gigabit speeds. Assume that the amount of IDS high speed memory is limited to storing say 1 Mbit. Then, assuming 100 bits per source, the IPS can only store information about a burst of 10,000 sources pending their transmission to a remote disk. This does not include the size of the Bloom filter, which can only store around 100,000 sources if scaled to 1 Mbit of size; after this point, the false positive rate starts increasing significantly. In practice one has to share the memory between the sources and the Bloom filter.

The inclination would be to clear the Bloom filter after it became full and start a second phase of logging. One concern is that timing synchronization could result in the same sources that were logged in phase 1 being logged and filling up the Bloom filter again, and this could happen repeatedly, leading to missing several sources. Even without this potential problem, there is danger in using a Bloom filter, as we can see by again considering the random model.

Consider enhancing the naïve logger with a Bloom filter to prevent the sending of duplicates. We assume the Bloom filter has a counter to track the number of items placed in the filter, and the filter is cleared when the counter reaches a threshold F to prevent too many false positives. Between each clearing, we obtain a group of F distinct random keys, but keys may be appear in multiple groups. Effectively, this generalizes the naïve logger, which simply used groups of size $F = 1$.

Not surprisingly, this variation of the coupon collector's problem has been studied; it is know as the coupon subset collection problem, and exact results for the problem are known [11, 14]. Details can be examined by the interested reader. A simple analysis, however, shows that for reasonable filter sizes F, there will be little or no gain over the naïve logger. Specifically, suppose $F = o(\sqrt{N})$. Then in the random model, the well-known birthday paradox implies that with high probability the first F keys to be placed in the Bloom filter will be distinct. While there may still be false positives from the Bloom filter, for such F the filter fills without detecting any true duplicates with high probability. Hence, in the random case, the expected collection time even using a Bloom filter of this size is still $\frac{N}{b}\ln(N - M) + O(1)$. With larger filters, some true duplicates will be suppressed, but one needs very large filters to obtain a noticeable gain. The essential point of this argument remains true even in the setting considered above where different sets of sources arrive at different speeds.

The key problem here is that we cannot supply the IDS with the list of all the sources that have been logged, even using a Bloom filter or a hashed set of fingerprints. Indeed, when $M << N$ no data structure can track a meaningful fraction of the keys that have already been stored to disk. Our solution to this problem is to partition the population of keys to be recorded into subsets of the right size, so that the logger can handle each subset without problem. The logger then iterates through all subsets in *phases*, as we now describe. This repeated cycling through the keys is reminiscent of a Carousel, yielding our name for our algorithm.

4 Scalable logging using Carousel

4.1 Partitioning and logging

Our goal is to partition the keys into subsets of the right size, so that during each phase we can concentrate on a single subset. The question is how to perform the partitioning. We want the size of each partition to be the right size for our logger memory, that is approximately size M. We suggest using a randomized partition of the sources into subsets using a hash function that uses very little memory and processing. This randomized partitioning would be simple if we initially knew the population size N, but that generally will not be the case; our system must find the current population size N, and indeed should react as the population size changes.

We choose a hash-based partition scheme that is particularly memory and time-efficient. Let $H(X)$ be a hash function that maps a source key X to an r-bit integer. Let $H_k(X)$ be the lower order k bits of $H(X)$. The size of the partition can be controlled by adjusting k.

For example, if $k = 1$, we divide the sources into two subsets, one subset whose low order bit (after hashing) is 1, and one whose lower order bit is a 0. If the hash function is well-behaved, these two sets will be approximately half the original size N. Similarly, $k = 2$ partitions the sources approximately into four equally sized subsets whose hash values have low order bits 00, 01, 10, and 11 respectively. This allows only very coarse-grained partitioning, but that is generally suitable for our purposes, and the simplicity of using the lower order k bits of $H(X)$ is particularly compelling for implementation and analysis. To begin we will assume the population size is stable but unknown, in which case the basic Carousel algorithm can be outlined as follows:

- *Partition:* Partition the population into groups of size 2^k by placing all sources which have the same value of $H_k(X)$ in the same partition.

- *Iterate:* A phase is assigned time $T_{phase} = M/b$ which is the time to log M sources, where M is the available memory in keys and b is the logging time. The i-th phase is defined by logging only sources such that $H_k(s) = i$. Other sources are automatically dropped during this phase. The algorithm must also utilize some means of preventing the same source from being logged multiple times in the phase, such as a Bloom filter or hash fingerprints.

- *Monitor:* If during phase i, the number of keys that match $H_k() = i$ exceeds a high threshold, then we return to the Partition step and increase k. While our algorithms typically use $k = k + 1$, higher jumps can allow faster response. If the number of number of keys that match $H_k() = i$ falls below a low threshold, then we return to the Partition step and decrease k.

In other words, Carousel initially tries to log all sources without hash partitioning. If that fails because of memory overflow, the algorithm then works on half the possible sources in a phase. If that fails, it works on a quarter of the possible sources, and so on. Once it determines the appropriate partition size, the algorithm iterates through all subsets to log all sources.

As described, we could in the monitoring stage change k by more than 1 if our estimate of the number of keys seen during that phase suggests that would be an appropriate choice. Also, of course, we can choose to decrease k if our estimate of the keys in that phase is quite small, as would happen if we are logging suspected virus sources and these sources are stopped. There are many variations and optimizations we could make, and some will be explored in our experiments. The important idea of Carousel, however, is to partition the set of keys to match the logger memory size, updating the partition as needed.

4.2 Collection Times for Carousel

We assume that the memory includes, for each key to be recorded, the space for the key itself, the corresponding report, and some number of bits for a Bloom filter. This requires slightly more memory space that we assumed when analyzing the random model, where we did not use the Bloom filter. The discrepancy is small, as we expect the Bloom filter to be less than 10% of the total memory space (on the order of 10 bits or less per item, against 100 or more bits for the key and report). This would not effectively change the lower bounds on performance of the naïve logger. We generally ignore the issue henceforth; it should be understood that the Bloom filter takes a small amount of additional space.

Recall that Carousel has 3 components: partition, iterate, and monitor. Faced with an unknown population N, the scalable logger will keep increasing the number of bits chosen k until each subset is less than size M, the memory size available for buffering logged keys.

We sketch an optimistic analysis, and then correct for the optimistic assumptions. Let us assume that all N keys are present at the start of time, that our hash function splits the keys perfectly equally, and that there is no failed recording of keys due to false positives from the Bloom filter (or whatever structure suppresses duplicates). In that case it will take at most $\lceil \log_2 \frac{N}{M} \rceil$ partition steps for Carousel to get the right number of subsets. Each such step required time for a single logging phase, $T_{phase} = M/b$. The logger then reaches the right subset size, so that k is the smallest value such that $N/2^k \leq M$. The collector then goes through 2^k phases to collect all N sources. Note that $2^k \leq 2N/M$, or else k would not be the smallest value with $N/2^k \leq M$. Hence, after the initial phases to find the right value of k, the additional collection time required is just $2N/b$, or a factor of two more than optimal. The total time is thus at most

$$\frac{M\lceil \log_2(N/M)\rceil}{b} + \frac{2N}{b},$$

and the generally the second term will dominate the first. Asymptotically, when $N >> M$, we are roughly within a factor of 2 of the optimal collection time.

Note that the factor of 2 in the $2N/b$ term could in fact be replaced in theory by any constant $a > 1$, by increasing the number of sets in the partition by a factor of a rather than 2 at each partition step. This would increase the number of partition steps to $\lceil \log_a \frac{N}{M} \rceil$. In practice we would not want to choose a value of a too close to 1, because keys will not be partitioned equally into sets, as we describe in the next subsection. Also, as we have described a factor of 2 is convenient in terms of partitioning via the low order bits of a hash. In what follows we continue to use the factor 2 in describing our algorithm, although it should be understood smaller constants (with other tradeoffs) are possible.

In some ways our analysis is actually pessimistic. Early phases that fail can still log some items, and we have assumed that we could partition to require $2N/M$ phases, when generally the number of phases required will be smaller. However, we have also made some optimistic assumptions that we now revisit more carefully.

4.2.1 Unequal Partitioning: Maximum Subset Analysis

If the logger uses k bits to partition keys, then there are $K = 2^k$ subsets. While the expected number of sources in a subset is $\frac{N}{K}$, even assuming a perfectly random hash function, there may be deviations in the set sizes. Our algorithm will actually choose the value of k such that the biggest partition is fit in our memory budget M, not the average partition, and we need to take this into account. That is, we need to analyze the *maximum* number of keys being assigned to a subset at each phase interval.

In general, this can be handled using standard Chernoff bound analysis [8]. In this specific case, for example, [10] proves that with very high probability, the maximum number of sources in any subset is less than $\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}}$. Therefore we can assume that the smallest integer k satisfying

$$\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}} \leq M, \tag{1}$$

where $K = 2^k$, is greater than or equal to the k eventually found by the algorithm.

Note that the difference between our optimistic analysis, where we required the smallest k such that $N/K \leq M$, and this analysis is generally very small, as $\sqrt{\frac{2N\ln K}{K}}$ is generally much less than N/K. That is, suppose that $N/K \leq M$, but $\frac{N}{K} + \sqrt{\frac{2N\ln K}{K}} > M$, so that at some point we might increase the value k to more than the smallest value such that $N/K \leq M$, because we unluckily have a subset in our partition that is bigger than the memory size. The key here is that in this case $N/K \approx M$, or more specifically

$$M \geq \frac{N}{K} > M - \sqrt{\frac{2N\ln K}{K}},$$

so that our collection time is now

$$\frac{2KM}{b} < \frac{2N}{b} + \frac{2}{b}\sqrt{\frac{2N\ln K}{K}}.$$

That is, the collection time is still, at most, very close to $2N/b$, with the addition of a smaller order term that contributes negligibly compared to $2N/b$ for large N. Hence, asymptotically, we are still with a factor of c of the optimal collection time, for any $c > 2$.

4.2.2 Effects of False Positives

So far, our analysis has not taken into account our method of suppressing duplicates. One natural approach is to use a Bloom filter, in which case false positives can lead to a source not being logged in a particular phase. This explains our definition of an (ϵ, c)-scalable logger. We have already seen that c can be upper bounded by any number larger than 2 asymptotically. Here ϵ can be bounded by the false positive rate of the corresponding Bloom filter. As long as the number of elements per phase is no more than $M' = \frac{N}{K} + \sqrt{\frac{2N\ln K}{K}}$ with high probability, then given the number of bits used for our Bloom filter, we can bound the false positive rate. For example, using $10M'$ bits in the Bloom filter, the false positive rate is less than 1%, so our logger asymptotically converges to a $(0.01, 2)$-scalable logger.

We make note of some additions one can make to improve the analysis. First, this analysis assumes only a single *major cycle* that logs each subset in the partition once. If one rerandomized the chosen hash functions each major cycle, then the probability a persistent source is missed each major cycle is independently at most ϵ each time. Hence, after two such cycles, the probability of a source being missed is at most ϵ^2, and so on.

Second, this analysis is pessimistic, in that in this setting, items are gradually added to an empty Bloom filter each phase; the Bloom filter is not in its full state at all times, so the false positive probability bound for the full filter is a large overestimate. For completeness we offer the following more refined analysis (which is standard) to obtain the expected false positive rate. (As usual, the actual rate is concentrated around its expectation with high probability.)

Assume the Bloom filter has m bits and uses h hash functions. Consider whether the $(i+1)$st item added to the filter causes a false positive. First consider a particular bit in the Bloom filter. The probability that it is not set to 1 by one of the hi hash functions thus far is $(1 - \frac{1}{m})^{hi}$. Therefore the probability of a false positive at this stage is $(1 - (1 - \frac{1}{m})^{hi})^h \approx (1 - e^{-\frac{hi}{m}})^h$.

Suppose M' items are added into the Bloom filter within a phase interval. The expected fraction of false positives is then (approximately) $\sum_{i=0}^{M'-1}(1 - e^{-\frac{hi}{m}})^h$, compared to the $(1 - e^{-\frac{hM'}{m}})^h$ given by the standard analysis for the false positive rate after M' elements have been added. As an example, with $M' = 312$, $h = 5$, and $m = 5000$, the standard analysis gives a false positive rate of $1.4 \cdot 10^{-3}$, while our improved analysis gives a false positive rate of $2.5 \cdot 10^{-4}$.

Third, if collecting all or nearly all sources is truly paramount, instead of using a Bloom filter, one can use hash-based fingerprints of the sources instead. This requires more space than a Bloom filter ($\Theta(\log M')$ bits per source if there are M' per phase) but can reduce the probability of a false positive to inverse polynomial in M'; that is, with high probability, all sources can be collected. We omit the standard analysis.

4.2.3 Carousel and Dynamic Adaptation

Under our persistent source assumption, any distinct key keeps arriving at the logger. In fact, for our algorithm as described, we need an even stronger assumption: each key must appear during the phase in which it is recorded, which means each key should arrive every N/b steps. Keys that do not appear this frequently may miss their phase and not be recorded. In most settings, we do not expect this to be a problem; any key that does not persist and appear this frequently does not likely represent a problematic source in terms of, for example, virus outbreaks. Our algorithm could be modified for this situation in various ways, which we leave as future work. One approach, for example, would be to sample keys in order to estimate the 95% percentile for average interarrival times between keys, and set the time interval for the phase time to gather a subset of keys accordingly.

A more pressing issue is that the persistent source assumption may not hold because external actions may shut down infected sources, effectively changing the size of the set of keys to record dynamically. For example, during a worm outbreak, the number of infected sources rises rapidly at first but then they can go down due to external actions (for example, network congestion, users shutting down slow machines due to infection, and firewalling traffic or blocking a

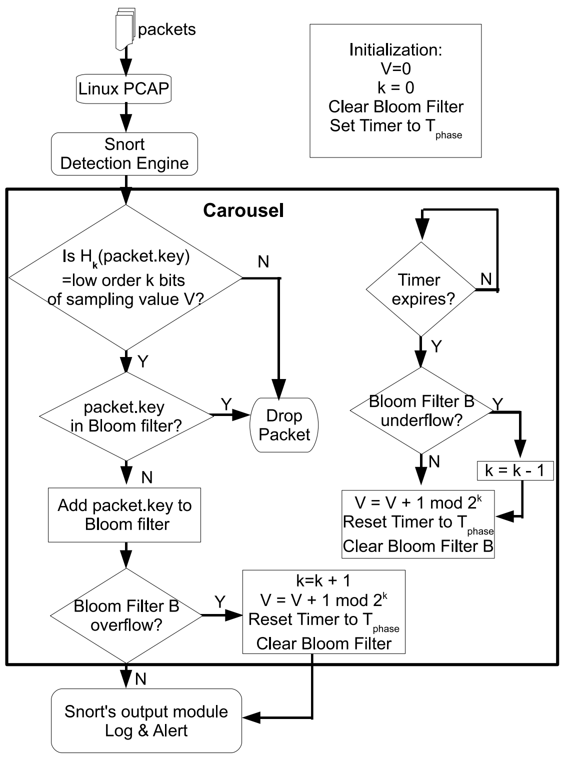

packets
Linux PCAP
Snort Detection Engine
Initialization: V=0 k = 0 Clear Bloom Filter Set Timer to T phase
Carousel
Is H k(packet.key) =low order k bits of sampling value V?
N
Y
packet.key in Bloom filter?
Y
N
Drop Packet
Add packet.key to Bloom filter
Bloom Filter B overflow?
Y
N
k=k + 1 V = V + 1 mod 2k Reset Timer to T phase Clear Bloom Filter
Timer expires?
N
Y
Bloom Filter B underflow?
Y
N
k = k - 1
V = V + 1 mod 2k Reset Timer to T phase Clear Bloom Filter B
Snort's output module Log & Alert

Figure 6: Flowchart of Carousel within Snort packet flow

part of the network). In that case, the scalable logger may pick a large number of sampling bits k at first due to large outbreak traffic. However, the logger should correspondingly increase the value of k subsequently as the number of sources to record declines, to avoid inefficient logging based on too large a number of phases.

5 Carousel Implementations

We describe our Snort evaluation in Section 5.1 and a sketch of a hardware implementation in Section 5.2.

5.1 Snort Implementation

In this section, we describe our implementation of Carousel integrated into the Snort [13] IDS. We need to first understand the packet processing flow within Snort to see where we can interpose the Carousel scalable logger scheme. As in Figure 6, incoming packets are captured by *libpcap*, queued in a kernel buffer, and then processed by the callback function *ProcessPacket*.

ProcessPacket first passes the packet to preprocessors, which are components or plug-ins serving to filter out suspicious activity and prepare the packet to be further analyzed. The detection engine then matches the packet against the rules loaded during Snort initialization. Finally, the Snort output module performs appropriate actions such as logging to files or generating alerts. Note that Snort is designed to be strictly single-threaded for multiplatform portability.

The logical choice is to place Carousel module between the detection engine and output module so that the traffic can either go directly to the output plugin or get diverted through the Carousel module. We cannot place the logger module before the detection engine because we need to log only after a rule (e.g., a detected worm) is matched. Similarly, we cannot place the logger after the output module because by then it is too late to affect which information is logged. Our implementation also allows a rule to bypass Carousel if needed and go directly to the output module.

Figure 6 is a flowchart of Carousel module for Snort interposed between the detection engine and the output model. The module uses the variables $T_{phase} = M/b$ (time for each phase) and k (number of sampling bits) described in Section 4.1. M is the number of keys that can be logged in a partition and b is the logging rate; in our experiments we use $M = 500$. The module also uses a 32-bit integer V that represents the hash value corresponding to the current partition. Initially, $k = 0$, $V = 0$, the Bloom filter is empty, and a timer T is set to fire after T_{phase}. The Bloom filter uses 5000 bits, or 10 bits per key that can fit in M, and employs 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [9].

The Carousel scalable logger first compares the low-order k bits of the hash of the packet key (we use the IP source address in all our experiments) to the low order k bits of V. If they do not match, the packet is not in the current partition and is not passed to the output logging. If the value matches but the key yields a positive from the Bloom filter (so it is either already logged, or a false positive), again the packet is not passed to the output module. If the value matches and the key does not yield a positive from the Bloom filter, then the module adds the key to the Bloom filter. If the Bloom filter overflows (the number of insertions exceeds M), then k is incremented by 1, to create smaller size partitions.

When the timer T expires, a phase ends. We first check for underflow by testing whether the number of insertions is less than M/x. We found empirically that a factor $x = 2.3$ worked well without causing oscillations. (A value slightly larger than 2 is sensible, to prevent oscillating because of the variance in partition sizes.) If there is no underflow, then the sampling value V is increased by 1 mod 2^k to move to the next partition.

5.2 Hardware Implementation

Figure 7 shows a schematic of the base logic that can be inserted between the detector and the memory buffer used to store log records in an IPS ASIC. Using 1 Mbit for the Bloom filter, we estimate that the logic takes less than 5% of a low-end 10mm by 10 mm networking ASIC. All re-

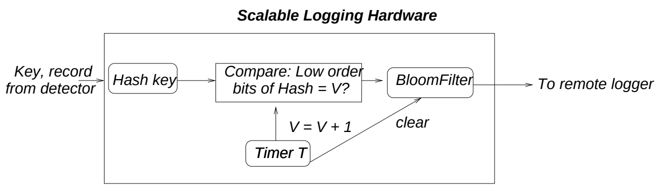

Scalable Logging Hardware
Key, record from detector
Hash key
Compare: Low order bits of Hash = V?
BloomFilter
To remote logger
V = V + 1
clear
Timer T

Figure 7: Schematic of the Carousel Logger logic as part of an IPS Chip.

sults are reported for a standard 400 Mhz 65 nm process currently being used by networking vendors. The logic is flow-through: in other words, it can inserted between the detector and logging logic without changing any other logic. This allows the hardware to be incrementally deployed *within* an IPS without changing existing chip sets.

We assume the detector passes a key (e.g., a source IP address) and a detection record (e.g., signature that matched) to the first block. The hash blocks computes a 64-bit hash of the key. Our estimates use a Rabin hash whose loop is unrolled to run at 40 Gbps using 20K gates.

The hash output supplies a 64-bit number which is passed to the Compare block. This block masks out the low-order k bits of the hash (a simple XOR) and then compares it (comparator) to a register value V that denotes the current hash value for this phase. If the comparison fails, the log attempt is dropped. If it succeeds, the key and record are passed to the Bloom filter logic. This is the most expensive part of the logic. Using 1 Mbit of SRAM to store the Bloom filter and 3 parallel hash functions (these can be found by taking bits 1-20, 21-40, 41-60 etc of the first 64-bit hash computed without any further hash computations), the Bloom filter logic takes less than a few percent of a standard ASIC.

As in the Snort implementation, a periodic timer module fires every $T_{phase} = M/b$ time and causes the value V to be incremented. Thus the remaining logic other than the Bloom filter (and to a smaller extent the hash computation) is very small. We use two copies of the Bloom filter and clear one copy while the other copy is used in a phase. The Bloom filter should be able to store a number of keys equal to the number of keys that can be stored in the memory buffer. Assuming 10 bits per entry, a 1 Mbit Bloom filter allows approximately 100,000 keys to be handled in each phase with the targeted false positive probability. Other details (underflow, overflow etc.) are similar to the Snort implementation and are not described here.

6 Simulation Evaluation

To evaluate Carousel under more realistic settings in which the population grows, we simulate the logger behavior when faced with a typical worm outbreak as modeled by a logistic

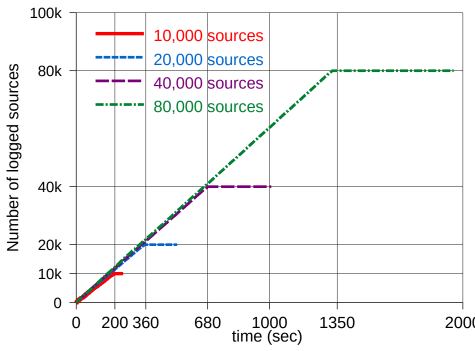


Figure 8: Performance of Carousel with different logging populations

equation. We used a discrete event simulation engine that is a stripped down (for efficiency) version of the engine found in ns-2. We implement the Carousel scalable logger as described in Section 4. The simulated logger maintains the sampling bit count k and only increases k when the Bloom filter overflows; k stabilizes when all sources sampled during T_{phase} fit the into memory budget M with logging speed b. Simulation allows us to investigate the effect of various input parameters such as varying worm speed and whether the worm uses a hit list. Again, in all the simulations below, the Bloom filter uses 5000 bits and 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [9]. For each experiment, we plot the average of 50 runs of simulation.

We start by confirming the theory with a baseline experiment in Section 6.1 when all sources are present at time 0. We examine the performance of our logger with the logistic model in Section 6.2. We evaluate the impact of non-uniform source arrivals in Section 6.3. In Section 6.4, we examine a tradeoff between using a smaller number of bits per Bloom filter element and taking more more major cycles to collect all sources. Finally, in Section 6.5, we demonstrate the benefit of reducing k in the presence of worm remediation.

6.1 Baseline Experiment

In Figure 8, we verify the underlying theory of Carousel in Section 4 assuming all sources are present at time 0. We consider various starting populations $N = 10000$ to 80000 sources, a memory budget of $M = 500$ items, and a logging speed $b = 100$ items per second.

Figure 8 shows that the Carousel scalable logger collects *almost all* (at least 99.9%) items by $t = 189, 354, 679$ and 1324 seconds for $N = 10000, 20000, 40000$ and 80000 respectively. This is no more than $\frac{2N}{b}$ in all cases, matching the predictions of our optimistic analysis in Section 4.

With these settings, the $10,000$ sources will be partitioned into 32 subsets, each of size approximately 312 (in expectation). In fact, our experiment trace shows that the number of sources per phase is in the range of 280 to 340. Since the Bloom filter uses 5000 bits, essentially we have more than 10 bits per item once the right number of partitions is found. As we calculated previously (in Section 4.2.2), the accumulated false positive rate of 312 sources in a 5000-bit Bloom filter with 5 hash functions is $2.5 \cdot 10^{-4}$. We also verified that most phases have no false positives. However, the Carousel algorithm may need additional major cycles to collect these remaining sources. Since a major cycle is 2^k iterations, the theory predicts that Carousel requires more time to collect missed false positives for larger k and hence for larger N. We observe that the length of horizontal segment of each curve in Figure 8, which represents the collection time of all sources missed in the first major cycle, is longer for larger populations N.

6.2 Logger Performance with Logistic Model

In the logistic model, a worm is characterized by H, the size of the initial hit list, the scanning rate, and a probability p of a scan infecting a vulnerable node. In our simulations below, we use a population of $N = 10,000$, a memory size $M = 500$ with Bloom filter and $M = 550$ without Bloom filter, and logging speed $b = 100$ packets/sec; the best possible logging time to collect all sources is $N/b = 100$ seconds.

For our first 3 experiments, shown in Figures 9, 10 and 11, we use an initial hit list of $H = 10,000$. Since the hit list is the entire population, as in the baseline, all sources are infected at time $t = 0$. We use these simulations to see the effect of increasing the scan rate and monitoring ability assuming all sources are infected. Our subsequent experiments will assume a much smaller hit list, more closely aligned with a real worm outbreak.

For the first experiment, shown in Figure 9 we use 6 scans per second (to model a worm outbreak that matches the Code Red scan rate [17]) and $p = 0.01$. Figure 9 shows that Carousel needs 200 seconds to collect the $N = 10,000$ sources whereas the naïve logger takes $4,000$ seconds. Further, the difference between Carousel and the naïve logger increases with the fraction of sources logged. For example, Carousel is 6 times faster at logging 90% level of all sources but 20 times faster to log 100% of all sources. This is consistent with the analysis in Section 3.1.

In Figure 10 we keep all the same parameters but increase the scan rate ten times to 60 scans/sec. The higher scan rate allows naïve logging a greater chance to randomly sample packets and so the difference between scalable and naïve logging is less pronounced. Figure 11 uses the same parameters as Figure 9 but assumes that only 50% of the scanning packets are seen by the IPS. This models the fact that a given IPS may not see all worm traffic. Notice again that the difference between naïve and Carousel logging decreases when the amount of traffic seen by the IPS decreases.

The remaining simulations assume a logistic model of worm growth starting with a hit list of $H = 10$ infected sources when the logging process starts. The innermost curve illustrates the infected population versus time, which obeys the well-known logistic curve. Even under this propagation model, Carousel still outperforms naïve logging by a factor of almost 5. Carousel takes around 400 seconds to collect all sources while naïve logger takes 2000 seconds.

Figure 13 shows a slower worm. A slower worm can be modeled in many ways, such using a lower initial hit list, a lower scan rate, or a lower victim hitting probability. In Figure 13, we used a smaller hitting probability of 0.001. Intuitively, the faster the propagation dynamics, the better the performance of the Carousel scalable logger when compared to the naïve logger. Thus the difference is less pronounced.

Figure 14 demonstrates the scalability of Carousel, as we scale up N from $10,000$ to $100,000$ with all other parameters staying the same (i.e., 6 scans per second and $p = 0.01$). Carousel takes around 9,000 seconds to collect all sources, while the naïve logger takes 40,000 seconds. Note also that in all simulations with the logistic model (and indeed in all our experiments) the performance of the naïve logger with a Bloom filter is indistinguishable from that of the naïve logger by itself — as the theory predicts.

6.3 Non-uniform source arrivals

In this section, we study logging performance when the sources arrive at different rates as described in Section 3.1. In particular, we experiment with two equal sets of sources in which one set sends at ten times as fast as the other set. Figure 15b shows the result for the naïve logger. We observe that the naïve logger has a significant problem in logging the slow sources, which are responsible for dragging down the overall performance. As predicted by our model, the times taken to log all slow sources is ten times slower than the time taken to log all fast sources. The times to log all and almost all sources are $8,000$ and $4,000$ seconds respectively.

Simply adding a Bloom filter only slightly increases the performance of the naïve logger as predicted by the theory . On the other hand, Carousel is able to consistently log all sources as shown in Figure 15a. Carousel is not susceptible to source arrival rates: sources from both the fast and slow sets are logged equally in each minor cycle once the appropriate number of sampling bits has been determined.

6.4 Effect of Changing Hash Functions

In this section, we study the effect of randomly changing the hash functions for the Bloom filter on each major cycle (that is, each pass through all of the sets of the partition). Recall that this prevents similar arrival patterns between major cycles from causing the same source to be missed repeatedly.

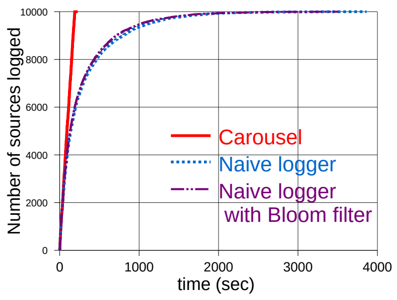


Figure 9: Performance of the Carousel scalable logger. Scan rate = 6/s, victim hit=1%, $M = 500$, $N = 10,000$, $b = 100$

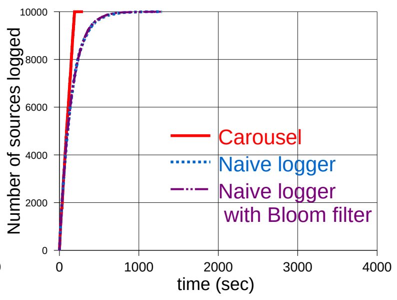


Figure 10: High scan rate (60 scans/s)

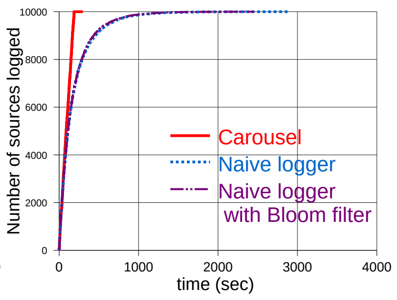


Figure 11: Reduced monitoring space (50%)

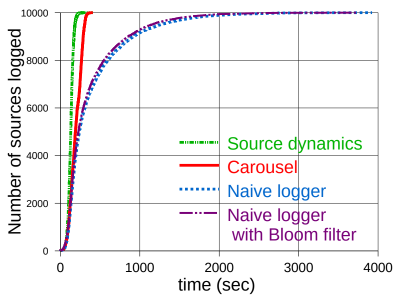


Figure 12: Logistic model of propagation - fast worm

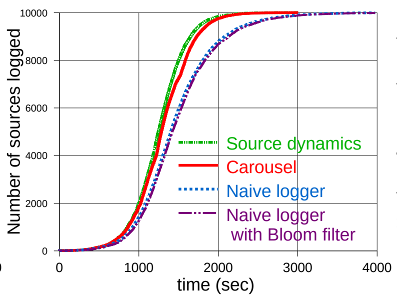


Figure 13: Logistic model of propagation - slow worm

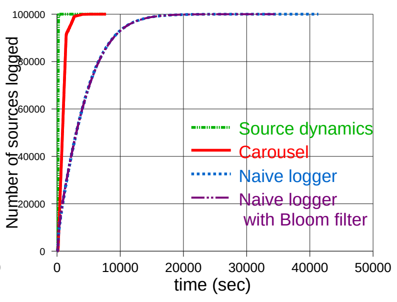


Figure 14: Scaling up the vulnerable population

Figure 17abc compares the performance in Carousel of using fixed hash functions throughout and changing the hash functions each major cycle with 1-bit, 5-bit and 10-bit Bloom filters respectively. We changed the hash functions randomly by simply XORing each hash value with a new random number after each major cycle. In these experiments, a major cycle is approximately 160 seconds. For the 1-bit results, one can clearly see knees in the curves at $t = 160$, 320, and 480 corresponding to each major cycle in which the logger collects sources missed in previous cycles.

Carousel instrumented with changing hash functions is much faster in collecting *all sources* across several major cycles. For example, for the 1-bit case, with changing hash functions each major cycle, it takes 1500 seconds to log all sources while using fixed hash functions takes 2500 seconds to log all sources.

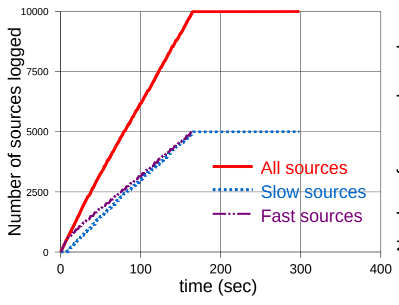
Should one prefer using a smaller number of bits per Bloom filter element and a greater number of major cycles or using a larger number of Bloom filter elements? This depends on the exact goals; for a fixed amount of memory, using a smaller number of Bloom filter bits per element allows the logger to log slightly more keys in every phase at the cost of a somewhat increased false positive probability. Based on our experiments, we believe using 5 bits per element provides excellent performance, although our Snort implementation (built before this experiment) currently uses 10 bits per element.

6.5 Adaptively Adjusting Sampling Bits

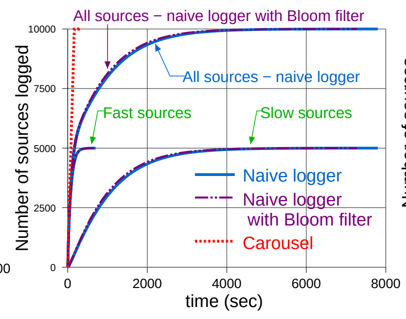
As described in Section 4.2, an optimization for Carousel is to dynamically adapt the number of sampling bits k to match the currently active source population. In a worm outbreak, the value of k needs to be large as the when the population of infected sources is large, but it should be decreased when the scope of the outbreak declines.

To study this effect, we use the *two-factor worm model* [17] to model the dynamic process of worm propagation coexisting with worm remediation. The two-factor worm model augments the standard worm model with two realistic factors: dynamic countermeasures by network administrators/users (such as node immunization and traffic firewalls) and additional congestion due to worm traffic that makes scan rates reduce when the worm grows. The model was validated using measurements of actual Internet worms (see [17]).

In Figure 16, we apply the two-factor worm model. The curve labeled “Source dynamics” records the number of infected sources as time progresses. Observe the exponential increase in the number of infected sources prior to $t = 100$.



(a) Carousel



(b) Naive logger

Figure 15: Logger performance under non-uniform source arrivals

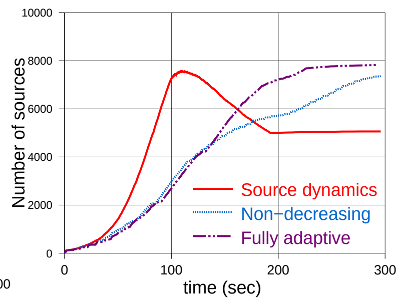
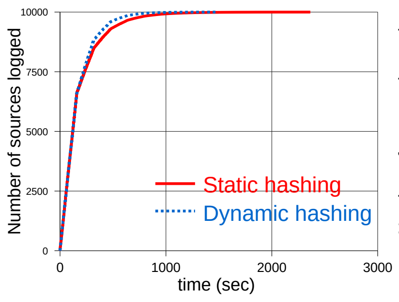
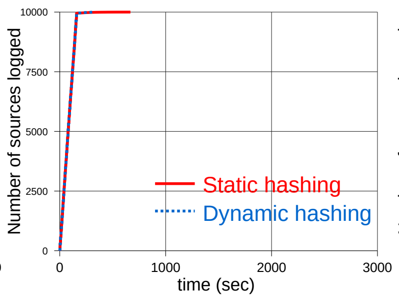


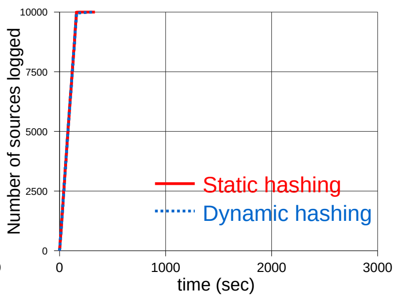
Figure 16: Dynamic source sampling in Carousel



(a) 1-bit Bloom filter



(b) 5-bit Bloom filter



(c) 10-bit Bloom filter

Figure 17: Comparison of fixed vs. changing hash functions in Carousel

However, the infected population then starts to decline.

If we let the two-factor model run to completion, the number of infected sources will eventually drop to zero, which makes logging sources less meaningful. In practice, however, it is the logging that makes remediation possible. Thus to illustrate the efficacy of using fully adaptive sampling within the logger, we only apply the two-factor model until the infectious population drops to half of the initial vulnerable tally. We then look at the time to collect the final infected population. Note that a non-decreasing logger will choose a sampling factor based on the peak population and thus may take unnecessarily long to collect the final population of infected sources.

Figure 16 shows that the fully adaptive scheme (increment k on overflow, decrement on underflow) enhances performance in terms of logging time and also the capability to collect more sources before they are immunized. In particular, the fully adaptive scheme collects almost all sources at 220 seconds while the non-decreasing scheme (only increments k on overflow, no decrements) takes more than 300 seconds to collect all sources. Examining the simulation results more closely, we found the non-decreasing scheme adapted to $k = 5$ (32 partitions) and stayed there, while the fully adaptive scheme eventually reduced to $k = 4$ (16 partitions) at time $t = 130$.

7 Snort Evaluation

We evaluate our implementation of Carousel in Snort using a testbed of two fast servers (Intel Xeon 2.8 GHz, 8 cores, 8 GB RAM) connected by a 10 Gbps link. The first server sends simulated packets to be logged according to a specified model while the second server runs Snort, with and without Carousel, to log packets.

We set the timer period $T_{phase} = 5$ seconds. The vulnerable population is $N = 10,000$ sources and the memory buffer has $M = 500$ entries. In the first experiment, the pattern of traffic arrival is random: each incoming packet is assigned a source that is uniformly and randomly picked from the population of N sources.

Figure 18 shows the logging performance of Snort instrumented with Carousel. Traffic arrives at the rate (B) of 100 Mbps. All packets have a fixed size of 1000 bytes. The logging rate is $b = 100$ events per second, i.e., $b \approx 1$ Mbps and $\frac{B}{b} = 100$. Figure 18 shows the improvements in logging from our modifications. Specifically, our scalable implementation is able to log all sources within 300 seconds

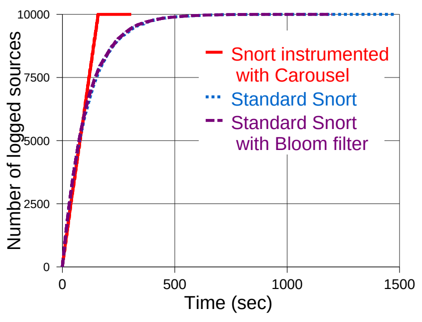


Figure 18: Logging performance of Snort instrumented with Carousel under a random traffic pattern

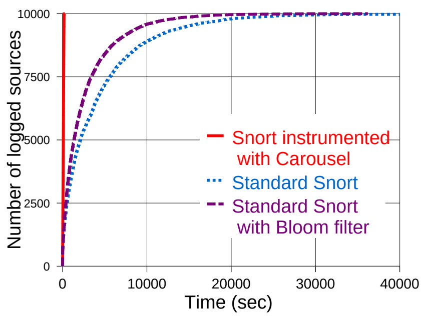


Figure 19: Logging performance of Snort instrumented with Carousel under a periodic traffic pattern

while standard Snort needs 1500 seconds. Also, adding a Bloom filter does not significantly improve the performance of Snort, matching our previous theory.

Figure 19 shows the logging performance when the sources are perpetually dispatched in a periodic pattern 1, 2, ..., N, 1, 2..., N, ... Such highly regular traffic patterns are common in a number of practical scenarios, such as synchronized attacks or periodic broadcasts of messages in the communication fabric of large distributed systems. We observe that the performance of standard Snort degrades by one order of magnitude as compared to the random pattern shown in Figure 18. Further examination shows that the naïve logger keeps missing certain sources due to the regular timing of the source arrivals. On the other hand, Carousel performance remains consistent in this setting.

We also performed an experiment with two equally sized sets of sources arriving at different rates, with fast sources arriving at 1 Gbps and slow sources at 100 Mbps, as shown in Figure 20. Our observations are consistent with the simulation results in Section 6.3. Note that in this setting standard Snort takes about 20 times longer to collect all sources than Snort with Carousel (300 seconds versus 6000 seconds); in contrast, Snort took only about 5 times longer in our experiment with random arrivals.

8 Related Work

A number of recent papers have focused on high speed implementations of IPS devices. These include papers on fast reassembly [4], fast normalization [15, 16], and fast regular expression matching (e.g., [12]). To the best of our knowledge, we have not seen prior work in network security that focuses on the problem of scalable logging. However, network managers are not just interested in detecting whether an attack has occurred but also in determining which of their computers is already infected for the purposes of remediation and forensics.

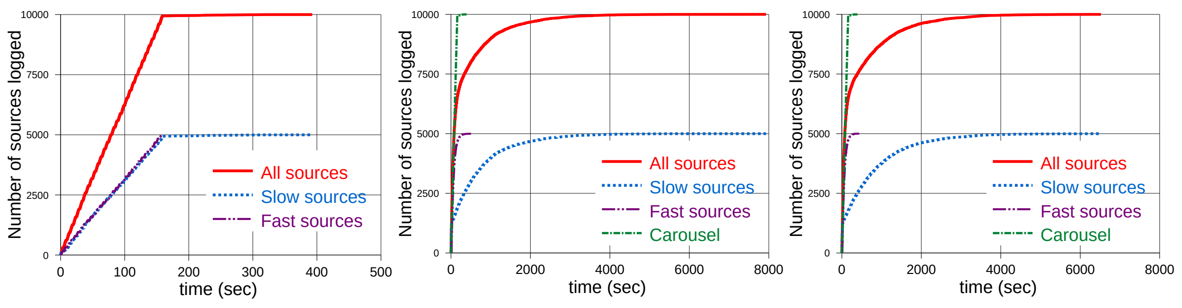
The use of random partitions, where the size is adjusted dynamically, is probably used in other contexts. We have found a reference to the Alto file system [6], where if the file system is too large to fit into memory (but is on disk), then the system resorts to a random partition strategy to rebuild the file index after a crash. Files are partitioned randomly into subsets until the subsets are small enough to fit in main memory. While the basic algorithm is similar, there are differences: we have *two* scarce resources (logging speed and memory) while the Alto algorithm only has one (memory). We have duplicates while the Alto algorithm has no duplicate files; we have an analysis, the Alto algorithm has none.

9 Conclusions

In the face of internal attacks and the need to isolate parts of an organization, IPS devices must be implementable cheaply in high speed hardware. IPS devices have successfully tackled hardware reassembly, normalization, and even Reg-Ex and behavior matching. However, when an attack is detected it is also crucial to also detect who the attacker was for potential remediation. While standard IPS devices can log source information, the slow speed of logging can result in lost information. We showed a naïve logger can take a multiplicative factor of $\ln N$ more time than needed, where N is the infected population size, for small values of memory M required for affordable hardware.

We then described the Carousel scalable logger that is easy to implement in software or hardware. Carousel collects nearly all sources, assuming they send persistently, in nearly optimal time. While large attacks such as worms and DoS attacks may be infrequent, the ability to collect a list of infected sources and bots without duplicates and loss seems like a useful addition to the repertoire of functions available to security managers.

While we have described Carousel in a security setting, the ideas applies to other monitoring tasks where the sources of all packets that match a predicate must be logged in the



(a) Carousel **(b) Standard Snort** **(c) Standard Snort with Bloom filter**

Figure 20: Snort under non-uniform source arrivals

face of high incoming speeds, low memory, and small logging speeds. The situation is akin to congestion control in networks; the classical solution, as found in say TCP or Ethernet, is for sources to reduce their rate. However, a passive logger cannot expect the sources to cooperate, especially when the sources are attackers. Thus, the Carousel scalable logger can be viewed as a form of randomized admission control where a random group of sources is admitted and logged in each phase. Another useful interpretation of our work is that while a Bloom filter of size M cannot usefully remove duplicates in a population of $N >> M$, the Carousel algorithm provides a way of recycling a small Bloom filter in a principled fashion to weed out duplicates in a very large population.

Acknowledgments

We thank the data center group of CSE Department at UCSD for the hardware setup.

Michael Mitzenmacher is supported in part by NSF grants CCF-0915922 and CNS-0721491, and from grants from Cisco Systems Inc., Yahoo! Inc., and Google Inc. Vinh The Lam and George Varghese are supported in part by grants from NSF and Cisco Systems Inc.

References

[1] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

[2] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. In *Cambridge University Press*, 1998.

[3] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Math*, 1(4), 2003.

[4] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *14th USENIX Security Symposium*, pages 5–5, 2005.

[5] S. Hogg. Security at 10 Gbps: http://www.networkworld.com/community/node/39071. In *Network World*, 2009.

[6] B Lampson. Alto: A personal computer. In *Computer Structures: Principles and Examples*, 1979.

[7] F. Li and et al. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), 2000.

[8] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[9] Arash Partow. General purpose hash functions: http://www.partow.net/programming/hashfunctions/.

[10] M. Raab and A. Steger. Balls into bins: a simple and tight analysis. In *Workshop on Randomization and Approximation Techniques in Computer Science*, 1998.

[11] A. Ross. The coupon subset collection problem. In *Journal of Applied Probability*, 2001.

[12] R. Smith and et al. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.*, 38(4):207–218, 2008.

[13] Snort. Snort ids: http://www.snort.org.

[14] W. Stadje. The collector's problem with group drawings. In *Advances Applied Probability*, 1990.

[15] G. Varghese, J. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. *SIGCOMM*, 36(4), 2006.

[16] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust tcp stream normalization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 96–110, 2008.

[17] C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *CCS '02*, pages 138–147. ACM, 2002.