Automated Worm Fingerprinting

Sumeet Singh, Cristian Estan, George Varghese and Stefan Savage
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Network worms are a clear and growing threat to the security of today's Internet-connected hosts and networks. The combination of the Internet's unrestricted connectivity and widespread software homogeneity allows network pathogens to exploit tremendous parallelism in their propagation. In fact, modern worms can spread so quickly, and so widely, that no human-mediated reaction can hope to contain an outbreak.

In this paper, we propose an automated approach for quickly detecting previously unknown worms and viruses based on two key behavioral characteristics – a common exploit sequence together with a range of unique sources generating infections and destinations being targeted. More importantly, our approach – called "content sifting" – automatically generates *precise* signatures that can then be used to filter or moderate the spread of the worm *elsewhere* in the network.

Using a combination of existing and novel algorithms we have developed a scalable content sifting implementation with low memory and CPU requirements. Over months of active use at UCSD, our *Earlybird* prototype system has automatically detected and generated signatures for all pathogens known to be active on our network as well as for several *new* worms and viruses which were *unknown* at the time our system identified them. Our initial experience suggests that, for a wide range of network pathogens, it may be practical to construct fully automated defenses – even against so-called "zero-day" epidemics.

1 Introduction

In the last three years, large-scale Internet worm outbreaks have *profoundly* demonstrated the threat posed by self-propagating programs. The combination of widespread software homogeneity and the Internet's unrestricted communication model creates an ideal climate for infectious pathogens. Worse, each new epidemic has demonstrated increased speed, virulence or sophistication over its predecessors. While the Code Red worm took over fourteen hours to infect its vulnerable population in 2001, the Slammer worm, released some 18 months later, did the same in under 10 minutes [22, 21]. The Code Red worm is thought to have infected roughly 360,000 hosts, while, by some estimates, the Nimda worm compromised over two million [8]. While early worms typically spread by a single mechanism and did little else, modern variants such as SoBig.F and MyDoom can spread through multiple vectors and have added backdoors, mail-relays and denial-of-service attacks to their payloads.

Unfortunately, our current ability to defend against these outbreaks is extremely poor and has not advanced significantly since the Code Red episode in mid-2001. In fact, the basic approach of detection, characterization, and containment has not changed significantly over the last five years. Typically, new worms are detected in an ad hoc fashion by a combination of intrusion detection systems and administrator legwork. Then, after isolating an instance of the worm, skilled security professionals manually characterize a worm *signature* and finally, this signature is used to contain subsequent infections via updates to anti-virus software and network filtering products. While this approach is qualitatively sound, it is quantitatively insufficient. Manual signature extraction is an expensive, slow, manual procedure that can take hours or even days to complete. It requires isolating a new worm, decompiling it, looking for invariant code sequences and testing the signature for uniqueness. However, recent simulations by Moore et al. suggest that an effective worm containment can require a reaction time of well under sixty seconds [23]. More concretely, consider that in the time it took to read this section, the Slammer worm had contacted well over a **billion** distinct Internet hosts.

This paper investigates the challenges in addressing this problem and describes a prototype system, called Earlybird, that can automatically detect and contain new worms on the network using precise signatures. Our approach, which we call *content sifting*, is based on two observations: first, that some portion of the content in existing worms is invariant – typically the code exploiting a latent host vulnerability – and second, that the spreading dynamics of a worm is atypical of Internet applications. Simply stated, it is rare to observe the same string recurring within packets sent from many sources to many destinations. By sifting through network traffic for content strings that are both frequently repeated and widely dispersed, we can automatically identify new worms and their precise signatures.

In our prototype system, we have developed approximate versions of this algorithm that are amenable to high-speed implementation. In live experiments on a portion of the UCSD campus network, we have deployed Earlybird and demonstrated that it can automatically extract the signature for all known active worms (e.g. CodeRed,

Slammer). Moreover, during our experiments Earlybird detected and extracted a signature for the Blaster, MyDoom and Kibuv.B worms – significantly before they had been publicly disclosed and hours or days before any public detection signatures were distributed. Finally, in our testing over a period of eight months, we have experienced relatively few false positives and exceptions are typically due to structural properties of a few popular protocols (SPAM via SMTP and NetBIOS) that recur consistently and can be procedurally "white-listed".

The remainder of this paper is structured as follows. In Section 2 we survey the field of worm research that we have built upon and describe how it motivates our work. Section 3 describes how we define worm behavior. Section 4 outlines a naive approach to detecting such behaviors, followed by a concrete description of practical content sifting algorithms in Section 5. Section 6 describes the implementation of the Earlybird prototype and an analysis of our live experiments using it. We describe limitations and extensions in Section 7. Finally, in Section 8 we summarize our findings and conclude.

2 Background and Related Work

Worms are simply small programs. They spread by exploiting a latent software vulnerability in some popular network service – such as email, Web or terminal access – seizing control of program execution and then sending a copy of themselves to other susceptible hosts.

While the potential threat posed by network worms has a long past – originating with fictional accounts in Gerrold's "When Harlie was One" and Brunner's "Shockwave Rider" – it is only recently that this threat has enjoyed significant research attention. Fred Cohen first lay the theoretical foundations for understanding computer viruses in 1984 [4, 5], and the Internet worm of 1988 demonstrated that self-replication via a network could dramatically amplify the virulence of such pathogens [33, 39]. However, the analysis and understanding of network worms did not advance substantially until the CodeRed outbreak of 2001. In this section, we attempt to summarize the contemporary research literature – especially in its relation to our own work.

The first research papers in the "modern worm era" focused on characterizations and analyses of particular worm outbreaks. For example, Moore et al. published one of the first empirical analyses of the CodeRed worm's growth, based on unsolicited scans passively observed on an unused network [22]. Further, the authors estimated the operational "repair" rate by actively probing a subsample of the 360,000 infected sites over time. They found that, despite unprecedented media coverage, the repair rate during the initial outbreak averaged under 2 percent per day. This reinforces our belief that fully automated intervention is necessary to effectively manage worm outbreaks. Staniford et al.'s landmark paper anticipated the development of far faster worms and extrapolated their growth analytically [42] – foreshadowing the release of the Slammer worm in 2002. Moore et al. subsequently analyzed the Slammer outbreak and estimated that almost *all* of the Internet address space was scanned by the worm in under 10 minutes – limited only by bandwidth constraints at the infected sites [21]. This experience also motivates the need for fast and automated reaction times. Finally, based on these results, Moore et al. analyzed the engineering requirements for reactive defenses – exploring the tradeoffs between reaction time, deployment and the granularity of containment mechanisms (signature based vs. IP address based) [23]. Two of their key findings motivate our work.

First, they demonstrated that signature-based methods can be an order of magnitude more effective than simply quarantining infected hosts piecemeal. The rough intuition for this is simple: if a worm can compromise a new host with an average latency of x seconds, then an address based quarantine can must react more quickly than x seconds to prevent the worm from spreading. By contrast, a signature based system can, in principle, halt all subsequent spreading once a signature is identified. The second important result was their derivation, via simulation, of "benchmarks" for how quickly such signatures must be generated to offer effective containment. Slow-spreading worms, such as CodeRed can be effectively contained if signatures are generated within 60 minutes, while containing high-speed worms, such as Slammer, may require signature generation in well under 5 minutes – perhaps as little as 60 seconds. Our principal contribution is demonstrating practical mechanisms for achieving this requirement.

In the remainder of this section we examine existing techniques for detecting worm outbreaks, characterizing worms and proposed countermeasures for mitigating worm spread.

2.1 Worm Detection

Three current classes of methods are used for detecting new worms: scan detection, honeypots, and behavioral techniques at end hosts. We consider each of these in turn.

Worms spread by selecting susceptible target hosts, infecting them over the network, and then repeating this process in a distributed recursive fashion. Many existing worms, excepting email viruses, will select targets using a random process. For instance, CodeRed selected target IP addresses uniformly from the entire address space. As a result, a worm may will be highly unusual in the number, frequency and distribution of addresses that it scans. This can be leveraged to detect worms in several ways.

To monitor random scanning worms from a global per-

spective, one approach is to use *network telescopes* – passive network monitors that observe large ranges of unused, yet routable, address space [25, 22, 26]. Under the assumption that worms will select target victims at random, a new worm will scan a given network telescope with a probability directly proportional to the worm's scan rate and the network telescope's "size"; that is, the number of IP addresses monitored. Consequently, large network telescopes will be able to detect fast spreading worms of this type fairly quickly. At the enterprise level, Staniford provides a comprehensive analysis of the factors impacting the ability of a network monitor to successfully detect and quarantine infected hosts in an online fashion [41].

However, there are two key limitations to the scan detection approach. First, it is not well suited to worms which spread in a non-random fashion, such as e-mail viruses or worms spread via instant messenger or peer-to-peer applications. Such worms generate a target list from address books or buddy lists at the victim and therefore spread *topologically* – according to the implicit relationship graph between individuals. Consequently, they do not exhibit anomalous scanning patterns and will not be detected as a consequence. The second drawback is that scan detection can only provide the IP address of infected sites, not a signature identifying their behavior. Consequently, defenses based on scan detection must be an order of magnitude faster than those based on signature extraction [23].

A different approach to worm detection is demonstrated by *Honeypots*. First introduced to the community via Cliff Stoll's book, "The Cuckoo's Egg", and Bill Cheswick's paper "An Evening with Berferd", honeypots are simply monitored idle hosts with untreated vulnerabilities. Any outside interaction with the host is, by definition, unsolicited and any malicious actions can be observed directly. Consequently, any unsolicited outbound traffic generated by a honeypot represents undeniable evidence of an intrusion and possibly a worm infection. Moreover, since the honeypot host is directly controlled, malicious code can be differentiated from the default configuration. In this manner, the "body" of a worm can be isolated and then analyzed to extract a signature. This approach is commonly used for acquiring worm instances for manual analysis [18]. There are two principal drawbacks to honeypots: they require a significant amount of slow manual analysis and they depend on the honeypot being quickly infected by a new worm.

Finally, a technique that has found increasing traction in the commercial world (e.g. via recently acquired startups, Okena and Entracept) is host-based behavioral detection. Such systems dynamically analyze the patterns of system calls for anomalous activity [31, 28, 3] indicating code injection or propagation. For example, attempts to send a packet from the same buffer containing a received packet is often indicative of suspicious activity. While behavioral techniques are able to leverage large amounts of detailed context about application and system behavior, they can be expensive to manage and deploy ubiquitously. Moreover, end-host systems can, by definition, only detect an attack against a single host and not infer the presence of a large-scale outbreak. Clearly, from a management, cost and reuse standpoint, it is ideal to detect and block new attacks in the network. That said, end-host approaches offer a level of sensitivity that is difficult to match in the network and can be a useful complement – particularly for detecting potential slow or stealthy worms that do not leave a significant imprint on the network.

2.2 Characterization

Characterization is the process of analyzing and identifying a new worm or exploit, so that targeted defenses may be deployed.

One approach is to create a priori *vulnerability signatures* that match known exploitable vulnerabilities in deployed software [44, 45]. For example, a vulnerability signature for the Slammer worm might match all UDP traffic on port 1434 that is longer than 100 bytes. By searching for such traffic, either in the network or on the host, a new worm exploiting the same vulnerability will be revealed. This is very similar to traditional intrusion detection systems (IDS), such as Snort [1] and Bro [29], which compare traffic content to databases of strings used in known attacks. This general approach has the advantage that it can deployed before the outbreak of a new worm and therefore can offer an added measure of defense. However, this sort of proactive characterization can only be applied to vulnerabilities that are already well-known and *well-characterized* manually. Further, the tradeoff between vulnerability signature specificity, complexity and false positives remains an open question. Wang et al's Shield, is by far the best-known vulnerability blocking system and it focuses on an end-host implementation precisely to better manage some of these tradeoffs [44]. We do not consider this approach further in this paper, but we believe it can be a valuable complement to the automated signature extraction alternative we explore.

The earliest automation for signature extraction is due to Kephart and Arnold [15]. Their system, used commercially by IBM, allows viruses to infect known "decoy" programs in a controlled environment, extracts the infected (i.e., modified) regions of the decoys and then uses a variety of heuristics to identify invariant code strings across infection instances. Among this set of candidates an "optimal" signature is determined by estimating the false positive probability against a measured corpus of

n-grams found in normal computer programs. This approach is extremely powerful, but assumes the presence of a known instance of a virus and a controlled environment to monitor.

The former limitation is partially addressed by the Honeycomb system of Kreibich and Crowcroft [17]. Honeycomb is a host-based intrusion detection system that automatically generates signatures by looking for longest common subsequences among sets of strings found in message exchanges. This basic procedure is similar to our own, but there are also important structural and algorithmic differences between our two approaches, the most important of which is scale. Honeycomb is designed for a host-based context with orders of magnitude less processing required. To put this in context, our Earlybird system currently processes more traffic in one second than the prototype Honeycomb observed in 24 hours. However, one clear advantage offered by the host context is its natural imperviousness to network evasion techniques [30]. We discuss this issue further in Section 7.

Finally, over the last two years of Earlybird's development [34, 35, 37], the clearest parallels can be drawn to Kim and Karp's contemporaneously-developed Autograph system [16]. Like Earlybird, Autograph also uses network-level data to infer worm signatures and both systems employ Rabin fingerprints to index counters of content substrings and use white-lists to set aside well-known false positives. However, there are several important differences as well. First, Autograph relies on a prefiltering step that identifies flows with suspicious scanning activity (particularly the number of unsuccessful TCP connection attempts) before calculating content prevalence. By contrast, Earlybird measures the prevalence of *all* content entering the network and only then considers the addressing activity. This difference means that Autograph cannot detect large classes of worms that Earlybird can – including almost all e-mail borne worms, such as MyDoom, UDP-based worms such as Slammer, spoofed source worms, or worms carried via IM or P2P clients. Second, Autograph has extensive support for distributed deployments – involving active cooperation between multiple sensors. By contrast, Earlybird has focused almost entirely on the algorithmics required to support a robust and scalable wire-speed implementation in a *single* sensor and only supports distribution through a centralized aggregator. Third, Earlybird is an on-line system that has been in near-production use for eight months and handles over 200 megabits of live traffic, while, as described, Autograph is an off-line system that has only been evaluated using traces. Finally, there are many differences in the details of the algorithms used (e.g. Autograph breaks content into non-overlapping variable-length chunks while Earlybird manages overlapping fixed-length content strings over each byte offset) although it is not currently clear what the impact of these differences is.

2.3 Containment

Containment refers to the mechanism used to slow or stop the spread of an active worm. There are three containment mechanisms in use today: host quarantine, string-matching and connection throttling. Host quarantine is simply the act of preventing an infected host from communicating with other hosts – typically implemented via IP-level access control lists on routers or firewalls. String-matching containment – typified by signature-based network intrusion prevention syst ems (NIPS) – matches network traffic against particular strings, or signatures, of known worms and can then drop associated packets. To enable high-bandwidth deployments, several hardware vendors are now producing high-speed string matching and regular expression checking chips for worm and virus filtering. Lockwood et al. describe an FPGA-based research prototype programmed for this application [19]. Finally, a different strategy, proposed by Twycross and Williamson [43], is to proactively limit the rate of all outgoing connections made by a machine and thereby slow – but not stop – the spread of any worm. Their approach was proposed in a host context, but there is no reason such connection throttling cannot be applied at the network level as well.

In this paper, we assume the availability of string-matching containment (perhaps in concert with throttling) and our Earlybird prototype generates signatures for a Snort in-line intrusion detection system – blocking all packets containing discovered worm signatures.

3 Defining Worm Behavior

Network worms, due to their distinct purpose, tend to behave quite differently from the popular client-server and peer-to-peer applications deployed on today's networks. In this section we explore these key behaviors in more detail and how they can be exploited to detect and characterize network worms.

3.1 Content invariance

In all existing worms of which we are aware, some or all of the worm program is invariant across every copy. Typically, the entire worm program is identical across every host it infects. However, some worms make use of limited polymorphism – by encrypting each worm instance independently and/or randomizing filler text. In these cases, much of the worm body is variable, but key portions are still invariant (e.g., the decryption routine). For the purposes of this paper, we assume that a worm has some amount of invariant content or has relatively few variants. We discuss violations of this assumption in Section 7.

3.2 Content prevalence

Since worms are designed foremost to *spread*, the invariant portion of a worm's content will appear frequently on the network as it spreads or attempts to spread. Conversely, content which is not prevalent will not represent the invariant portion of a worm and therefore is not a useful candidate for constructing signatures.

3.3 Address dispersion

For the same reasons, the number of distinct hosts infected by a worm will grow over time. Consequently, packets containing a live worm will tend to reflect a variety of different source and destination addresses. Moreover, during a major outbreak, the number of such addresses can grow extremely quickly. Finally, it is reasonable to expect that the distribution of these addresses will be far more uniform than typical network traffic which can have significant clustering [9].[1] In this paper we only take advantage of the first of these three observations, but we believe there is potential value in considering all of them.

4 Finding worm signatures

From these assumptions, we can conclude that network worms must generate significant traffic to spread and that this traffic will contain common substrings and will be directed between a variety of different sources and destinations. While it is not yet clear that this characterization is *exclusively* caused by worms, for now we will assume that identifying this traffic pattern is sufficient for detecting worms. We examine the issue of false positives later in the paper. In principle, detecting this traffic pattern is relatively straightforward. Figure 1 shows an idealized algorithm that achieves this goal. For each network packet, the content is extracted and all substrings processed. Each substring is indexed into a prevalence table that increments a count field for a given substring each time it is found. In effect, this table implements a histogram of all observed substrings. To maintain a count of unique source and destination addresses, each table entry also maintains two lists, containing IP addresses, that are searched and potentially updated each time a substring count is incremented. Sorting this table on the substring count and the size of the address lists will produce the set of likely worm traffic. Better still, the table entries meeting this worm behavior criteria are exactly those containing the invariant substrings of the worm. *It is these substrings that can be used as signatures to filter the worm out of legitimate network traffic.*

We call this approach *content sifting* because it effectively implements a high-pass filter on the contents of network traffic. Network content which is not prevalent or not widely dispersed is sifted out, leaving only the worm-like content. However, while content sifting can correctly identify worm signatures, the basic algorithm we have described is far too inefficient to be practical. In the next section we describe algorithms for approximating correctness in exchange for efficiency and practicality.

[1] As described earlier, the presence of "dark" IP addresses can also provide qualitatively strong evidence of worm-like behavior.

```
ProcessTraffic(payload,srcIP,dstIP)
1   prevalence[payload]++
2   Insert(srcIP,dispersion[payload].sources)
3   Insert(dstIP,dispersion[payload].dests)
4   if (prevalence[payload]>PrevalenceTh
5       and size(dispersion[payload].sources)>SrcDispTh
6       and size(dispersion[payload].dests)>DstDispTh)
7       if (payload in knownSignatures)
8           return
9       endif
10      Insert(payload,knownSignatures)
11      NewSignatureAlarm(payload)
12  endif
```

Figure 1: The idealized content sifting algorithm detects all packet contents that are seen often enough and are coming from enough sources and going to enough destinations. The value of the detection thresholds and the time window over which each table is used are both parameters of the algorithm.

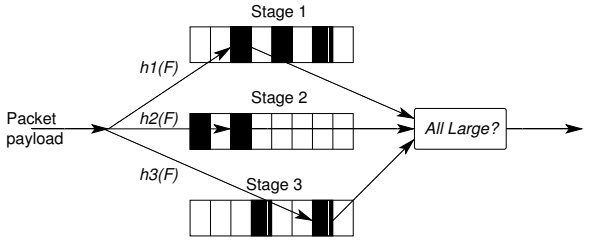

Figure 2: *Multi-stage Filters.* A piece of content is hashed using hash function $h1$ into a Stage 1 table, $h2$ into a Stage 2 table, etc and each table entry contains a counter that is incremented. If *all* the hashed counters are above the prevalence threshold, then the content string is saved for address dispersion measurements. In previous work we have shown that the probability of an approximation error decreases exponentially with the number of stages and consequently is extremely small in practice [10].

5 Practical Content Sifting

For automated signature extraction to scale to high-speed links, the algorithms must have small processing requirements (ideally well-suited to parallelization), and small memory requirements. Finally, to allow arbitrary deployment strategies, the algorithm should not *depend* on having a symmetric vantage point in the network. To satisfy these requirements, we now describe scalable and accurate algorithms for estimating content prevalence and address dispersion, and techniques for managing CPU overload through smooth tradeoffs between detection time and overhead. For simplicity, in this section we describe our algorithms at packet (and not flow) granularity.

5.1 Estimating content prevalence

Identifying common content involves finding the packet payloads that appear at least x times among the N packets sent during a given interval. However, a table indexed

by payload can quickly consume huge amounts of memory. For example, on a fully loaded 1 Gbps link, this naive approach could generate a 1 GByte table in less than 10 seconds. Memory consumption can be reduced considerably by indexing the table using a fixed size hash of the packet payload instead of the full payload. After a certain hash value has repeated $x - 1$ times, the next packet with this hash is reported. In the absence of collisions, the associated content will have appeared exactly x times. By selecting a hash function with suitably large range (e.g., 32 or 64 bits) the collision probability can be minimized. Assuming a 16 byte hash table entry and an average packet size of 500 bytes, this algorithm would take over 4 minutes to generate the same 1 GByte table.

Memory efficiency can be improved further by observing that identifying prevalent content is isomorphic to the well-studied problem of identifying high-bandwidth flows, frequently called "heavy hitters" [13, 10]. By modifying the definition of "flow" to reflect content fingerprints instead of the (*srcip, dstip, srcport, dstport, protocol*) tuple used for flow analysis, heavy-hitter approximation algorithms can be used to find prevalent content using comparatively small amounts of memory.

Our prototype uses *multi-stage filters* with conservative update to dramatically reduce the memory footprint of the problem (see Figure 2 for a general description and [13, 10] for a thorough analysis). While simple, we believe this notion of using a content signature as a "flow identifier" on which to maintain counters is a powerful technique.[2]

An important modification is to append the destination port and protocol to the content before hashing. Since worms typically target a particular service (they are designed to exploit a vulnerability in that service) this will not impact the ability to track worm traffic, but can effectively exclude large amounts of prevalent content not generated by worms (i.e., potential false positives).[3] For example, if two users on the same network both download the Yahoo home page they will receive many packets with identical payloads. However, traffic sent from the Web server will be directed to a so-called "ephemeral" port selected by each client. Since these ports are selected independently, adding them to the hash input will generally differentiate these different clients even when the content being carried is identical.

So far, we have only discussed content at the *whole* packet granularity. While this is sufficient for detecting most existing worms, it is easy to envision worms for which the invariant content is a string smaller than a single packet or for which the invariant content occurs at different offsets in each instance. However, detecting common strings of at least a minimum length is computationally complex. Instead we address the related – yet far easier – problem of detecting repeating strings with a small fixed length β. As with full packet contents, storing individual substrings can require exorbitant memory and computational resources. Instead, we use a method similar to the one proposed by Manber for finding similar files in a large file system [20]. We compute a variant of Rabin fingerprints for all possible substrings of a certain length [32]. As these fingerprints are polynomials they can be computed incrementally while retaining the property that two equal substrings will generate the same fingerprint, no matter where they are in a packet.

However, each packet with a payload of s bytes has $s - \beta + 1$ strings of length β, so the memory references used per packet is still substantially greater than that consumed by a single per-packet hash. In Section 5.3, we describe a technique called *value sampling* to considerably reduce memory references.

5.2 Estimating address dispersion

While content prevalence is the key metric for identifying potential worm signatures, address dispersion is critical for avoiding false positives among this set. Without this additional test a system could not distinguish between a worm and a piece of content that frequently occurs between two computers – for example a mail client sending the same user name repeatedly as it checks for new mail on the mail server regularly.

To quantify address dispersion one must count the *distinct* source IP addresses and destination IP addresses associated with each piece of content suspected of being generated by a worm (note that this is different from the previous content prevalence problem which only requires estimating the *repetitions* of each distinct string). While one could simply count source-destination address pairs, counting the source and destination addresses independently allows finer distinctions to be made. For example, mail messages sent to a popular mailing list are associated with many source-destination address pairs, but with only two sources – the mail server of the original sender of the message and the mail server running the list.

While it is possible to count IP addresses exactly using a simple list or hash table, more efficient solutions are needed if there are many pieces of content suspected of being generated by worms. Our solution is to trade off some precision in these counters for dramatic reductions in memory requirements. Our first approach was to appropriate the *direct bitmap* data structure originally

[2] We are not the first to use hashing techniques to analyze the content makeup of network traffic. Snoeren et al. and Duffield et al. both use hashing to match packet observations across a network [38, 7], and both Spring et al. and Muthitacharoen et al. use Rabin fingerprints for compressing content sent over a network [40, 27].

[3] Note that it is possible for this assumption to be violated under unusual circumstances. In particular, the Witty worm exploited promiscuous network devices and only required a fixed *source* port to exploit its vulnerability – the destination port was random [6]. Catching this worm required us to maintain an additional matching table in which the source port is appended to the hash output instead.

developed for approximate flow counting [46, 11]. Each content source is hashed to a bitmap, the corresponding bit is set, and an alarm is raised when the number of bits set exceeds a threshold. For example, if the dispersion threshold T is 30, the source address is hashed into a bitmap of 32 bits and an alarm is raised if the number of bits set crosses 20 (the value 20 is calculated analytically to account for hash collisions). This approach has minimal memory requirements, but in exchange it loses the ability to estimate the actual values of each counter – important for measuring the rate of infection or prioritizing alerts. While other techniques such as probabilistic counting [12] and multiresolution bitmaps [11] can provide accurate counts they require significantly more memory. For example a multiresolution bitmap requires 512 bits to count to 1 million.

Instead, we have invented a counting algorithm that leverages the fact that address dispersion continuously increases during an outbreak. Using this observation we devise a new, compact data structure, called a *scaled bitmap*, that accurately estimates address dispersion using five times less memory than existing algorithms.

The scaled bitmap achieves this reduction by subsampling the range of the hash space. For example, to count up to 64 sources using 32 bits, one might hash sources into a space from 0 to 63 yet only set bits for values that hash between 0 and 31 – thus ignoring half of the sources. At the end of a fixed measurement interval, this subsampling is adjusted by scaling the resulting count to estimate the true count (a factor of two in the previous example). Generalizing, we track a continuously increasing count by simply increasing this scaling factor whenever the bitmap is filled. For example the next configuration of the bitmap might map one quarter of the hash space to a 32 bit bitmap and scale the resulting count by four. This allows the storage of the bitmap to remain constant across an enormous range of counts.

However, once the bitmap is scaled to a new configuration, the addresses that were active throughout the previous configuration are lost and adjusting for this bias directly can lead to double counting. To minimize these errors, the final scaled bitmap algorithm, shown in Figure 3, uses multiple bitmaps ($numbmps = 3$ in this example) each mapped to progressively smaller and smaller portions of the hash space. To calculate the count, the estimated number of sources hashing to each bitmap are added, and then this sum is divided by the fraction of the hash space covered by all the bitmaps. When the bitmap covering the largest portion of the hash space has too many bits set to be accurate, it is advanced to the next configuration by recycling it: the bitmap is reset and then mapped to the next slice of the hash space (Figure 4). Consequently, each bitmap covers half the hash space covered by its predecessor. The first bitmap, the one covering the largest portion of the hash space, is the most important in computing the estimate, but the other bitmaps provide "memory" for counts that are still small and serve to minimize the previously mentioned biases. Consequently, not much correction is needed when these bitmaps become the most important. Formally, we can prove that the maximum ratio between the bias of the algorithm and the number of active addresses is $2/(2^{numbmps} - 1)$[36].

```
UpdateBitmap(IP)
1  code = Hash(IP)
2  level = CountLeadingZeroes(code)
3  bitcode = FirstBits(code << (level+1))
4  if (level ≥ base and level < base+numbmps)
5     SetBit(bitcode,bitmaps[level-base])
6     if (level == base and CountBitsSet(bitmaps[0]) == max)
7        NextConfiguration()
8     endif
9  endif

ComputeEstimate(bitmaps,base)
1  numIPs=0
2  for i= 0 to numbmps-1
3     numIPs=numIPs+b ln(b/CountBitsNotSet(bitmaps[i]))
4  endfor
5  correction=
          2(2^base − 1)/(2^numbmps − 1) · b ln(b/(b − max))
6  return numIPs ·2^base/(1 − 2^−numbmps)+correction
```

Figure 3: A scaled bitmap uses $numbmps$ bitmaps of size b bits each. The bitmaps cover progressively smaller portions of the hash space. When the bitmap covering the largest portion of the hash space gets too full to be accurate (the number of bits set reaches max), we advance to the next configuration by "recycling" the bitmap (see Figure 4). To compute an estimate of the number of distinct IP addresses, we multiply a estimate of the number of addresses that mapped to the bitmaps by the inverse of the fraction of the hash space covered by the bitmaps. A correction is added to the result to account for the IP addresses that were active in earlier configurations, while the current bitmaps were not in use at their present levels.

Overall, this new technique allows us to count sources and destinations quite accurately using only 3 bitmaps with roughly 5 times less memory than previously known techniques [12, 11]. This is critical for practical scaling because it reduces the system's sensitivity to the effectiveness of the low-pass filter provided by the content prevalence test.

5.3 CPU scaling

Using multistage filters to detect content prevalence and scaled bitmaps to estimate address dispersion decreases memory usage and limits the amount of processing. However, each payload string still requires significant processing. In our prototype implementation (detailed in Section 6), the CPU can easily manage processing each packet payload as a single string, but when applying Rabin fingerprints, the processing of every substring of length β can overload the CPU during high traffic load. For example, a packet with $1,000$ bytes of payload and $\beta = 40$, requires processing 960 Rabin fingerprints. While computing the Rabin fingerprints themselves incurs overhead, it is the three order of magnitude

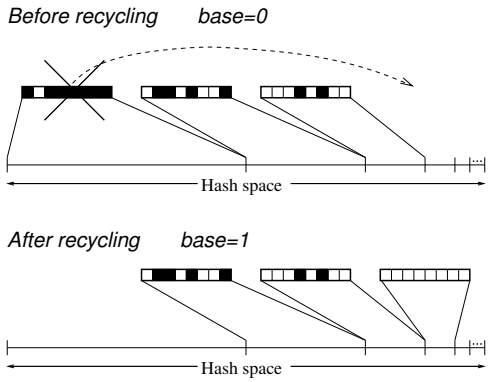

Figure 4: When the bitmap covering the largest portion of the hash space fills up, it is recycled. The bitmap is cleared and it is mapped to the largest uncovered portion of the hash space which is half the size of the portion covered by the bitmap rightmost before recycling. Recycling increments the variable $base$ (see Figure 3) by one.

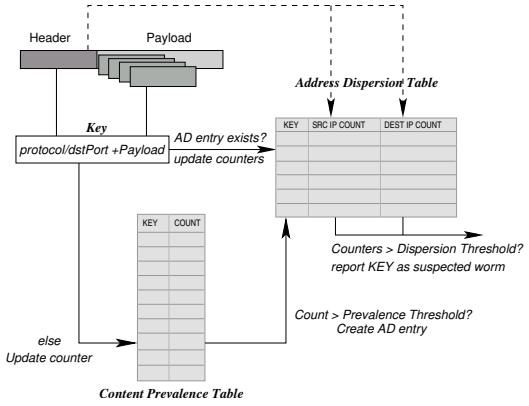

Figure 5: Content Sifting Algorithm as used in EarlyBird.

increase in the number of content sifting operations that exceeds the capacity of our current CPU. While a faster CPU might solve this problem for a given traffic profile, the possibility of traffic surges and denial-of-service attacks on a sensor produce the same problem again. We believe that a security device should not fail in these circumstances but instead smoothly scale back functionality to match capacity – still performing the same functions but perhaps with reduced fidelity or responsiveness.

The obvious approach to address this problem is via dynamic sampling. However, randomly sampling which substrings to process could cause us to miss a large fraction of the occurrences of each substring and thus delay the generation of a worm's signature. Instead, we use *value sampling* [20] and select only those substrings for which the fingerprint matches a certain pattern (e.g. the last 6 bits of the fingerprint are 0). Consequently, the algorithm will systematically ignore some substrings, but track all occurrences of others. However, if a worm contains even a single tracked substring, it will be detected as promptly as without the sampling. For example, if f is the fraction of the tracked substrings (e.g. $f = 1/64$ if we track the substrings whose Rabin fingerprint ends on 6 0s), then the probability of detecting a worm with a signature of length x is $p_{track}(x) = 1 - e^{-f(x-\beta+1)}$.

Since Rabin fingerprint are randomly distributed themselves, the probability of tracking a worm substring of length β is f. Thus, the probability of missing the worm is $p_{miss}(\beta) = 1 - f$. The probability of not tracking the worm is the probability of not tracking any of its substrings. If the worm signature has length x, it has $x - \beta + 1$ substrings of length β. Assuming that no substring of length β repeats in the signature, the probability of not tracking the worm is $p_{miss}(x) = (1-f)^{x-\beta+1} \approx e^{-f(x-\beta+1)}$. For example with $f = 1/64$ and $\beta = 40$, the probability of tracking a worm with a signature of 100 bytes is 55%, but for a worm with a signature of 200 bytes it increases to 92%, and for 400 bytes to 99.64%.

The sampling value f represents a tradeoff between processing and the probability of missing a worm; processing decreases linearly with f and the length of the invariant content required increases linearly with f. Note that all current worms have had invariant content of at least 400 bytes, for which the probability of false negatives is at most 0.36%. Our user-space software implementation requires $f = 1/64$ to keep up with roughly 200Mbps of traffic on a Gigabit Ethernet interface. Finally, since the parameters of the Rabin fingerprint algorithm p and M are not known, the worm writer cannot determine which strings will not be sampled in advance.

5.4 Putting it together

Figure 5 depicts the content sifting algorithm implemented in the EarlyBird prototype. As each packet arrives, its content (or substrings of its content) is hashed and appended with the protocol identifier and destination port to produce a content hash code. In our implementation, we use a 32-bit Cyclic Redundancy Check (CRC) as a packet hash and 40-byte Rabin fingerprints for substring hashes. Each Rabin fingerprint is subsampled with $f = 1/64$. The resulting hash codes are used to index the address dispersion table. If an entry already exists (the content has been determined to be prevalent) then the address dispersion table entries for source and destination IP addresses (implemented as scaled bitmaps) are updated. If the source and destination counts exceed the dispersion threshold, then the content string is reported.

If the content hash is not found in the dispersion table, it is indexed into the content prevalence table. In our implementation, we use four independent hash functions of the content hash to create 4 indexes into four counter arrays. Using the *conservative update* optimization, only the smallest among the four counters is incremented [10]. If all four counters are greater than the prevalence threshold, then a new entry is made in the ad-

dress dispersion table – with high probability, the content has appeared frequently enough to be a candidate worm signature. Pseudocode for the main loop of the EarlyBird system is shown in Figure 5.

```
ProcessPacket()
1  InitializeIncrementalHash(payload,payloadLength,dstPort)
2  while (currentHash=GetNextHash())
3     if (currentADEntry=ADEntryMap.Find(currentHash))
4        UpdateADEntry(currentADEntry,srcIP,dstIP,packetTime)
5        if ( (currentADEntry.srcCount > SrcDispTh)
               and (currentADEntry.dstCount > DstDispTh) )
6           ReportAnomalousADEntry(currentADEntry,packet)
7        endif
8     else
9        if ( MsfIncrement(currentHash) > PravalenceTh)
10          newADEntry=InitializeADEntry(srcIP,dstIP,packetTime)
11          ADEntryMap.Insert(currentHash,newADEntry)
12       endif
13    endif
14 endwhile
```

Figure 6: The EarlyBird loop performed on every packet.When the prevalence threshold is exceeded, dispersion counting is done by creating an ADentry. ADentry contains the source and destination bitmaps and the scale factors required for the scaled bitmap implementation.

The content prevalence table sees the most activity in the system and serves as a high-pass filter for frequent content. The multi-stage filter data structure is cleared on a regular interval (60 seconds in our implementation). By contrast, the address prevalence table has typically fewer values – only those strings exceeding the prevalence threshold – and can be garbage collected over longer time scales (even hours).

Each of these mechanisms can be implemented at high speeds in either software or hardware, with relatively modest memory requirements as we quantify in the next section. Moreover, our approach makes no assumptions about the point of deployment, whether at the endpoint, edge, or core. However the optimal parameters settings may depend on the point of deployments. In Section 6 we empirically explore the parameter settings used by our EarlyBird prototype.

6 Experience

Based on the content sifting algorithm just described, we have built a prototype system which has been in use on the UCSD campus for over eight months. In this section, we describe our overall system design, the implementation and experimental environment, our initial experiments exploring the parameter space of the content sifting algorithm, our evaluation of false positives and false negatives, and our preliminary results in finding live worms at our site.

6.1 System design

The EarlyBird system consists of two major components: *Sensors* and an *Aggregator*. Each sensor sifts through traffic on configurable address space "zones" of responsibility and reports anomalous signatures. The aggregator

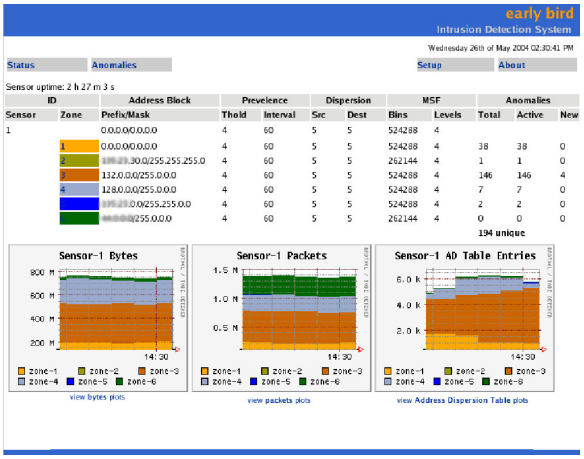

Figure 7: A screenshot of the main screen of the EarlyBird user interface. Each zone is labeled by a prefix and shows the current anomalies (worms), and prevalence/dispersion parameters which can be changed by the user. More detailed screens show detailed counts for each anomaly, as shown for Sasser in Figure 12.

coordinates real-time updates from the sensors, coalesces related signatures, activates any network-level or host-level blocking services and is responsible for administrative reporting and control. Our implementation is written in C and the aggregator also uses the MySql database to log all events, the popular rrd-tools library for graphical reporting, and PHP scripting for administrative control. A screenshot of the main screen of the EarlyBird user interface showing zones and a summary of the current system activity is shown in Figure 7.

Finally, in order to automatically block outbreaks, the EarlyBird system automatically generates and deploys precise content-based signatures formatted for the Snort-inline intrusion prevention system [1]. A sample such signature for Kibvu.B is shown below.

```
drop tcp $HOME_NET any -> $EXTERNAL_NET 5000
(msg:"2712067784 Fri May 14 03:51:00 2004";
rev:1; content:"|90 90 90 90 4d 3f e3 77 90
90 90 90 ff 63 64 90 90 90 90 90|";)
```

6.2 Implementation and environment

The current prototype Earlybird sensor executes on a 1.6Ghz AMD Opteron 242 1U server configured with a standard Linux 2.6 kernel. The server is equipped with two Broadcom Gigabit copper network interfaces for data capture. The EarlyBird sensor itself is a single-threaded application which executes at user-level and captures packets using the popular libpcap library. The system is roughly 5000 lines of code (not including external libraries) with the bulk of the code dedicated to self-monitoring for the purpose of this paper. The scalable implementation itself is a much smaller fraction of this code base. In its present untuned form, EarlyBird sifts though over 1TB of traffic per day and is able to keep up with over 200Mbps of continuous traffic when

using Rabin fingerprints with a value sampling probability of $1/64$ (and at even higher speeds using whole packet CRCs).

The experiments in the remainder of this paper are based on data collected from a Cisco Catalyst router configured to mirror all in-bound and out-bound traffic to our sensor (Earlybird currently makes no distinction between incoming and outgoing packets). The router manages traffic to and from roughly 5000 hosts, primarily clients, as well as all traffic to and from a few dedicated campus servers for DNS, SMTP/POP/IMAP, NFS, etc. The measured links experience a sustained traffic rate of roughly 100Mbps, with bursts of up to 500Mbps.

6.3 Parameter tuning

The key parameters used by our algorithm are the content prevalence threshold (currently 3), the address dispersion threshold (currently 30 sources and 30 destinations), and the time to garbage collect address dispersion table entries (currently several hours). We now describe the rationale behind these initial choices.

Content prevalence threshold: Figure 8 shows the distribution of signature repetitions on a trace for different hash functions. For example, using a 60 second measurement interval and a whole packet CRC, over 97 percent of all signatures repeat two or fewer times and 94.5 percent are only observed once. Using a finer grained-content hash or a longer measurement interval increases these numbers even further. However, to a first approximation, all reasonable values of these parameters reveal that very few signatures ever repeat more than 3 times. Recall that the principal benefit provided by the content prevalence table is to remove from consideration the enormous number of substrings which appear rarely and therefore are not possible worm signature candidates. We have repeated these experiments on several datasets at differing times and observed the same pattern. Consequently, for the remainder of this paper we use a prevalence threshold of 3.

Address dispersion threshold: Once a signature has passed the prevalence threshold it is still unlikely that it represents a worm. Figure 9 shows the number of distinct signatures found, as a function of time, for different address dispersion thresholds. For example, after 10 minutes there are over 1000 signatures (note the log scale) with a low dispersion threshold of 2 – meaning that the same string has been observed in packets with two different source IP addresses *and* two different destination IP addresses. However, as the dispersion threshold is increased, the number of such strings decreases dramatically. By the time a threshold of 30 is reached, there are only 5 or 6 prevalent strings meeting the dispersion criteria and the increase in this number is very slow over time. In this particular trace, two of these strings represent live

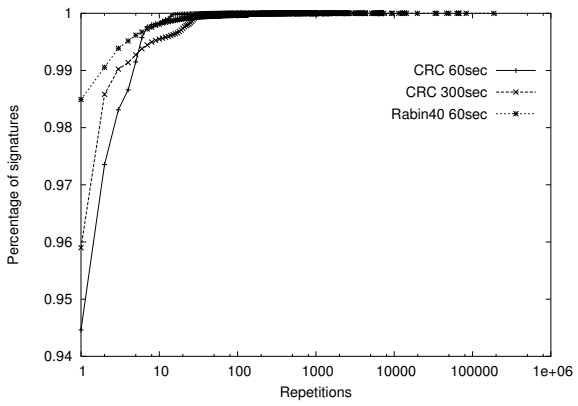

Figure 8: Cumulative distribution function of content signatures for different hash functions. This CDF is computed from the set of repetitions found in each measurement interval over a period of 10 minutes. Note that the y axis is artificially truncated to show more detail.

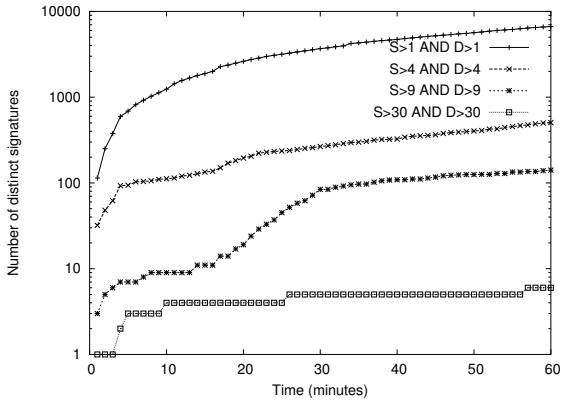

Figure 9: Number of distinct signatures detected over time for different address dispersion thresholds.

worms and the others are benign but consistently reoccurring strings that are post-filtered by a whitelist.

Note that there is an inherent tradeoff between the speed of detecting a new worm and the likelihood of a false positive. By using a lower dispersion threshold one can respond to a worm more quickly, but it is increasingly likely that many such signatures will be benign. For example, we find that the Slammer worm signature is detected within one second with an address dispersion threshold of 2, yet takes up to 5 seconds to discover using the more conservative threshold of 30. At the same time there are two orders of magnitude more signatures that will be reported with the lowest dispersion threshold – most of which will likely be false positives.

Garbage collection: The final key parameter of our algorithm is the elapsed time before an entry in the address dispersion table is garbage collected. The impact of this setting is shown in Figure 10. When the timeout is set to 100 seconds, then almost 60 percent of all signatures are garbage collected before a subsequent update

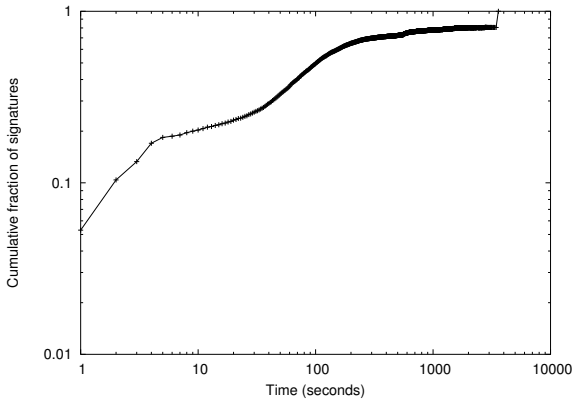

Figure 10: Log-scale cumulative distribution function of the maximum time period between updates for entries in the address dispersion table.

occurs – possibly preventing the signature from meeting the dispersion threshold and being reported. However, by a timeout of 1000 seconds, this number is reduced to roughly 20 percent of signatures. However, since the total number of signatures in the address dispersion table is always fairly small (roughly 25,000) we can comfortably maintain a timeout of several hours.

6.4 Performance

In Section 5 we described mechanisms such as multistage filters to reduce memory, and mechanisms such as value sampling to reduce CPU time. In this section, we briefly evaluate the performance of our prototype EarlyBird sensor in terms of processing time and memory.

Processing Time: To measure overhead, we instrumented the interface of each component to count elapsed CPU cycles. Because these counters are measured on a live system with varying packet sizes, and some functions (e.g., computing the Rabin hash) depend on packet size, we report the average and standard deviation over several million packets.

The top of Table 1 shows the overhead (in microseconds) incurred by each individual component of the EarlyBird algorithm as shown in Figure 6. The most significant operations are the initial Rabin fingerprint, accessing the multistage filter and creating a new Address Dispersion Table entry (dominated by the cost of malloc in this implementation). The Rabin algorithm is highly optimized based on Manber's original code (as modified by Neil Spring), and incrementally amounts to a multiply and a mask (AND) operation. It seems difficult to optimize this particular hash function further in software, but it would be easy to implement in hardware. Similarly, a hardware implementation of multistage filters can use parallel memories to reduce the lookup time in proportion to the number of filter stages.

The bottom of Table 1 shows the overall processing time (in microseconds) taken by EarlyBird to process a

	Mean	Std. Dev.
Component wise breakdown		
Rabin Fingerprint		
First Fingerprint (40 bytes)	0.349	0.472
Increment (each byte)	0.037	0.004
Multi Stage Filter		
Test & Increment	0.146	0.049
AD Table Entry		
Lookup	0.021	0.032
Update	0.027	0.013
Create	0.252	0.306
Insert	0.113	0.075
Overall Packet		
Header Parsing & First Fingerprint	0.444	0.522
Per-byte processing	0.409	0.148
Overall Packet with Flow-Reassembly		
Header Parsing & Flow maintenance	0.671	0.923
Per-byte processing	0.451	0.186

Table 1: This table shows overhead (in microseconds) incurred by each of the individual operations performed on a packet. The mean and standard deviation are computed over a 10 minute interval (25 million Packets). This table represents raw overheads before sampling. Using 1 in 64 value sampling, the effective mean per byte processing time reduces to 0.042 microseconds.

packet. Without the use of value sampling, on an average it takes approximately 0.44 microseconds to parse the packet header and compute the first hash from the packet payload. Additionally for each byte in the packet payload EarlyBird adds an additional processing overhead of 0.409 microseconds on average. Utilizing (1 in 64) value sampling, as described in Section 5, brings down the average per-byte time to under 0.042 microseconds. This equates to 0.005 microseconds per bit, or a 200 Mbps line rate. We confirmed this by also examining the packet drop rate and comparing the output packet rate of the router, and the input packet rate seen by the system: at a sampling rate of 1 in 64 there are almost no dropped packets during 200Mbps load, but at smaller sampling rates there were significant numbers of dropped packets for equivalent input. In hardware, given the same value sampling rate, assuming that multiplies can be pipelined, and that the multistage filter memories and address dispersion tables operate in parallel, there is no reason why the algorithm should not scale with memory speeds even up to 40 Gbps.

Memory Consumption: The major memory hog is the content prevalence table, implemented using multistage filters with 4 stages, with each stage containing 524288 bins, and where each bin is 8 bits, for a total of 2MB. While this number of bins may appear to be large, recall that we are using a small prevalence threshold, and the amount of memory is still dramatically smaller than what would be required to index all content substrings.

The other major component of memory usage is the Address Dispersion Table which, in our experience, has

between 5K and 25K entries of 28 bytes each. All 25,000 of the Address Dispersion table entries combined utilize well under a Megabyte of memory. The other components use a negligible amount of memory. Thus the core EarlyBird function currently consumes less than 4 Mbytes of memory. If the content prevalence threshold was made higher (as could be done in deployments nearer the core of the network), the memory needs of the multistage filters and address dispersion tables will go down dramatically, allowing potential on-chip implementations; current FPGAs allow 1 Mbyte of on-chip SRAM, and custom chips allow up to 32 Mbytes.

6.5 Trace-based verification

In this subsection, we report on experimental results for false positives and false negatives. While a key feature of our system is that it runs on *live* network traffic, for these experiments we replayed a captured trace in real-time to support comparisons across runs. We report on some of our live experience in the next subsection.

False Positives: Any worm detection device must contend with false positives. Figure 11 shows the prevalence of different signatures over time that meet the dispersion threshold of 10 (we set the threshold in this experiment lower than the current parameter setting of 30 used in our live system to produce more signatures), while the signatures themselves are listed in Table 2. The two most active signatures belong to the Slammer and Opaserv worms, followed by a pervasive string on TCP port 445 used for distributed port scanning, and the Blaster worm. The remaining signatures fall into two categories: those that are likely worms or distributed scans (likely, due to the high degree of such traffic sent to these ports from outside the LAN) and a few strings that represent true false positives and arise from particular protocol structures.

Over longer live runs we have found two principal sources of false positives: common protocol headers and unsolicited bulk email (SPAM). In the former category, over 99 percent of all false positives result from distinct SMTP header strings or HTTP user-agent or content-type strings. We have observed over 2000 of these combinations in practice, all of which are easily whitelisted procedurally via a protocol parser. It is critical to do so however, since automatically blocking or rate-limiting even a significant subset of HTTP traffic could have disastrous consequences. The other principal source of false positives are SPAM e-mails which can exceed address dispersion thresholds due to the use of distributed mailers and mail relays. While these are far more difficult to whilelist since many e-mail viruses also propagate via TCP port 25, the effect of their interdiction is far more benign as well. Moreover, false positives arising from SPAM are bursty in practice since they coincide with a mass mailing cycle. Consequently, even without additional policy, countermeasures for such traffic tends to automatically self-limit.

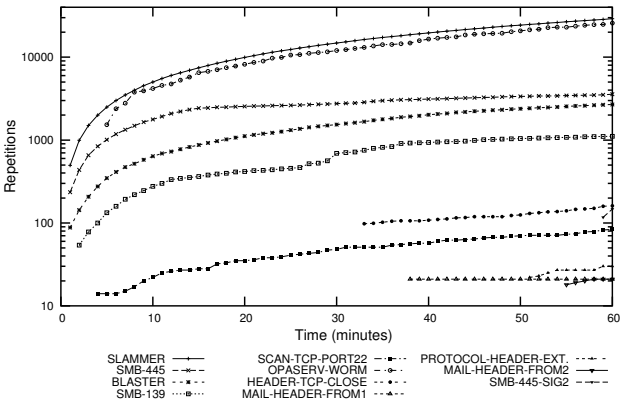

Figure 11: Count of worm-like signatures as a function of time. This graph uses a content prevalence threshold of 3 and an address dispersion threshold of 10 (i.e. $d > 10$ and $s > 10$).

We have also seen a few false positives that simply reflected unique aspects of a few popular protocol implementations. For example, the most common signatures in our data (not shown above) are strings of all zeros and all ones in base64 encoding (caused by a popular mail reader's text encoding) and we exclude these from consideration statically. Similarly, the HEADER-TCP-CLOSE signature show above, is a string "tcp-close, during connect" that is included in TCP RST packets sent by the MAC OS X operating system. The system itself is sufficiently common that the address dispersion and content prevalence criteria are met. However, there are sufficiently few of these, even over extended periods of analysis, that a handful of static whitelist entries have been sufficient to remove them.

Finally, over long traces we have observed one source of false positives that defies easy analysis. In particular, popular files distributed by the BitTorrent peer-to-peer system can satisfy the content prevalence and address dispersion criteria during their peak periods of popularity. This does not seem to happen in practice with other peer-to-peer systems that rely on whole-file download, but BitTorrent's file striping creates a many-to-many download profile that mimics that of a worm.

In general, all of our false positives appear consistently across multiple trials over different time periods. This leads us to believe that most are relatively stable and reflect a small number of pathologies. Consequently, in live use we excluded such signatures in a small "white list" which is used to post-filter signature reports.

False negatives: Since our experiments have been run in an uncontrolled environment it is not possible to quantitatively demonstrate the absence of false negatives. However, a strong qualitative indication is that Earlybird running live detected every worm outbreak reported on

Label	Service	Sources	Dests
SLAMMER	UDP/1434	3328	23607
SCAN-TCP-PORT22	TCP/22	70	53
MAIL-HEADER-FROM	TCP/25	12	11
SMB-139	TCP/139	603	378
SMB-445	TCP/445	2039	223
HEADER-TCP-CLOSE	TCP/80	33	136
MAIL-HEADER-FROM2	TCP/25	13	14
PROTOCOL-HEADER-EXT	TCP/80	15	24
BLASTER	TCP/135	1690	17
OPASERV-WORM	UDP/137	180	21033
SMB-445-SIG2	TCP/445	11	145

Table 2: Summary signatures reported using an address dispersion threshold of 10.

public security mailing lists (including BugTraq, Full-Disclosure, and snort-signatures) during our period of operation. We also checked for false negatives by comparing the trace we used against a Snort rulebase including over 340 worm and worm-related signatures aggregated from the official Snort distribution as well as the snort-signatures mailing list. We found no false negatives via this method, although the Snort system alerted on a number of instances that were not worms.

6.6 Inter-packet signatures

As described, the content-sifting algorithm used in EarlyBird does not keep any per-flow state and can therefore only generate content signatures that are fully contained within a single packet. Thus an attacker might evade detection by splitting an invariant string into pieces one byte smaller than β – one per packet.

We have extended the content sifting algorithm to detect such simple evasions at the cost of per flow state management. While there are many approaches to flow reassembly, our initial design point is one that trades the fully general reassembly for reduced overhead by exploiting Earlybird's use of incremental fingerprints. Thus, for each flow we maintain a circular buffer containing the last 40 bytes, as well as the Rabin fingerprint for this string and some book-keeping state for managing flow expiration. Using this data we are able to continue the signature matching process from each packet to its successor (in fact the computation cost is reduced as a result, since the expensive per-packet Rabin fingerprint is only necessary for the first packet in a flow). Currently, we manage flows first-come first-served via a flow cache, although it is easy to bias this allocation to favor sources exhibiting abnormal activity such as port-scanning or accessing unused address space [34, 35, 37, 24]. It should be clear that a sophisticated worm or virus which subverts the host operating system will be able to reorder or arbitrarily delay packets in a way that evades this approach. We describe the challenges of more complex evasions in Section 7. We briefly evaluated the performance impact of this extension and found that using a flow cache of 131072 elements (7MB in total) the average cost for processing the first packet of a flow is increased by 0.227 microseconds and the average per-byte cost is increased by 0.042 (absolute numbers and associated standard deviations are reported in Table 1).

6.7 Live experience with EarlyBird

In addition to the worms described above, Earlybird has also detected precise signatures for variants of CodeRed, the MyDoom mail worm and most recently for the Sasser, and Kibvu.B worm. In the case of new worms such as Kibvu.B and MyDoom, Earlybird reported signatures long before there were public reports of the worm's spread – let alone signatures available – and we were able to use these signatures to assist our network operations staff in tracking down infected hosts.

While we have experience with all recent worms, we limit ourselves to describing our experience with two recent outbreaks, Sasser and Kibvu.B.

Sasser: We detected Sasser on the morning of Saturday May 1st, 2004. Though we cannot claim to be the first ones to detect Sasser, we certanly did detect it before signatures were made available by the various anti-virus vendors. Part of the reason for us not detecting Sasser earlier is because all inbound traffic destined to port 445 is dropped at the upstream router and thus we could only use strictly internal traffic to make an identification. Figure 12 shows a screenshot of the live EarlyBird system tracking the rate in growth of infected Sasser hosts and their attempts to infect others in the UCSD network.

Kibvu.B: Kibvu.B is a recent worm that Earlybird detected on Friday May 14th, 2003 at 3:08AM PDT. In contrast to other outbreaks, Kibvu.B was extremely subdued (perhaps because it targeted a two year old vulnerability that was less prevalent in the host population). In the 40 minute window following our first recorded instance of the worm, there were a total of 37 infection attempts to 30 unique destinations, allowing us to trigger based on our 30-30 dispersion threshold. The Kibvu.B experience suggests that simply utilizing content prevalence as a metric as in [16] may not be sufficient; address dispersion is essential. We have provided a signature for this worm in section 6.1.

7 Limitations and Extensions

While we have been highly successful with our prototype system, we recognize a number of limitations, potential challenges and problems facing those building a complete worm defense system. In this section we discuss these and discuss current extensions that we are adding to our system to address these issues.

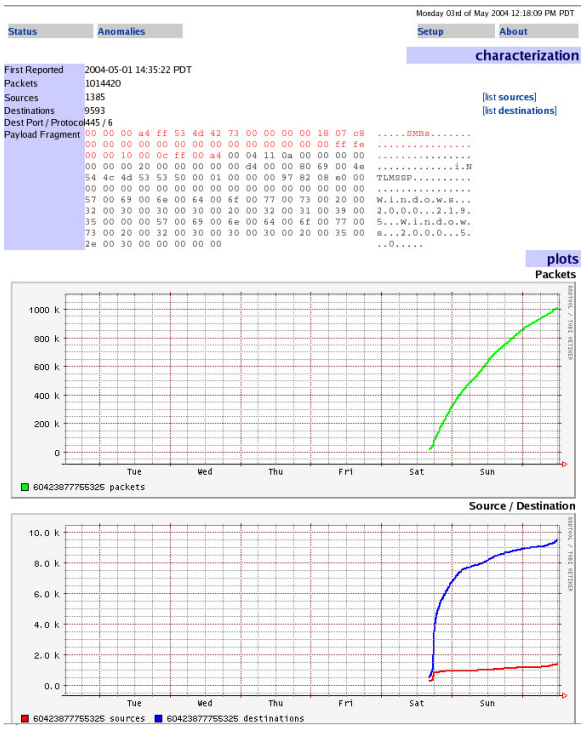

Figure 12: A detailed screen showing EarlyBird's capture of the Sasser outbreak. The anomaly is labeled 6042387755325 because at the time of discovery the anomaly was not named. The top of the screen-shot shows the signature and the number of sources and destinations. The middle of the screen shot shows a time-series plot of the packets containing the signature over roughly 2 days. The bottom of the screen shot shows a time-series plot of unique destinations (top curve) and unique sources (bottom curve) with the content. The destinations are much larger because the same sources are attempting to infect a large number of destinations.

7.1 Variant content

If content sifting were to be widely deployed this could create an incentive for worm writers to design worms with little or no invariant content. For example, polymorphic viruses encrypt their content in each generation and so-called "metamorphic viruses" have even demonstrated the ability to mutate their entire instruction sequence with semantically equivalent, but textually distinct, code. This is an extremely challenging problem that is currently addressed by antivirus vendors using controlled emulation [2] and procedural signatures. While many of these subterfuges are trivially detectable (e.g. since polymorphic decryption code may be itself invariant), and others can detected by modifying our content sifting approach to identify textually "similar" content – in the limit this threat is a fundamental one. As part of future work we are investigating hybrid pattern matching approaches that quickly separate non-code strings (identifiable by unavoidable terminating instruction sequences) from potential exploits – and focus complex analysis only on those sequences which pose a threat.

Other problems are presented by compression. While existing self-encoding viruses maintain an invariant decoding routine, a worm author might choose to reuse a common code sequence (e.g., such as one used for a popular self-decompressing executable format, like ZIP executables). Using this string as a worm signature could produce many false positives for content using the same sequence. Finally, several vulnerabilities have been found in popular implementations of encrypted session protocols such as ssh and SSL. Worms exploiting such vulnerabilities can *opportunistically* make use of the per-session encryption offered by these services. As a result, content-oriented traffic analysis, like that described in this paper, would be impossible. The same problem is posed by widespread deployment of end-to-end IPSEC and virtual private networks (VPNs). This problem appears to be fundamental. Indeed if such deployments become widespread much of the current security market (especially including current intrusion detection systems) will have to be rethought.

7.2 Network evasion

Just as attackers may attempt to evade content sifting algorithms by creating metamorphic worms, they may also attempt to evade our monitor through traditional IDS evasion techniques [30]. While we discussed the problem of *flow reassembly* earlier, a sophisticated attacker might send overlapping IP fragments or TCP segments to create a network-level polymorphism. To address this issue requires *traffic normalization*, in which datagrams are reassembled in a single consistent fashion [14]. However, in its full generality, this approach requires far more per-flow state and per-packet computation than mere flow reassembly and therefore may not scale well without further performance-enhancing techniques. An alternative we are considering is to simply filter such obviously oddball packets – at the cost of some impact on sites which actually depend on non-standard TCP segmentation or IP fragmentation implementations.

Finally, incidental network evasion may occur if the assumptions underlying the address dispersion threshold are violated. For example, if a worm requires only a single packet for transmission then the attacker could spoof the source address so all packets appear to originate from the same source. While such evasions are easy to detect, it is requires special purpose code outside the general content sifting framework.

7.3 Extensions

In addition to the potential challenges posed by malicious actors, there are a number of additional improvements that could be made to our system even in the current environment. For example, while we have experienced that given parameter settings appear to provide consistent results on our link across time, our settings were

themselves based on measurement and experimentation. We believe they are sensitive to the number of live hosts interdicted by the monitor, but exactly how remains an open question. In the next generation of our system, we plan to use techniques similar to [10] to "autotune" EarlyBird's content sifting parameters for a given environment.

Finally, while most worms to date have sought to maximize their growth over time, it is important to address the issue of slow worms as well. In our current prototype, worms which are seen less frequently than every 60 seconds have no hope of registering. One method to address this limitation within our system is to maintain triggering data across multiple time scales. Alternatively, one might deploy a hybrid system, using Earlybird to intercept high-speed outbreaks worms and host-based intrusion detection or large-scale honeypots to detect slowly spreading pathogens. Indeed, even small detection probabilities can eliminate the stealthy advantage of slow worms and thus the incentive for deploying them.

7.4 Containment

Our current system reports the suspected worm signatures, but can be configured to generate Snort signatures in a few seconds which can then be blocked by an online Snort deployment. We have been doing so on a small scale on a laboratory switch, and the system has blocked worm traffic based on the signatures we feed the blocker. Unfortunately, the policy for applying such a containment strategy can be quite complex. For example, since there is an inherent tradeoff between detection speed and false positives, as we discussed earlier, one reasonable policy is to temporarily rate-limit traffic matching signatures with only moderate address dispersion. If the signature is a false positive then it likely will never reach a higher level of dispersion and the rate-limit can be repealed. If it is a worm, then this conservative reaction will slow its spread and once its dispersion increases to a higher level the system can decide to drop all packets carrying the signature. However, this is just one such policy option and the question deserves additional attention.

Moreover, automated containment also provokes the issue of attackers purposely trying to trigger a worm defense – thereby causing denial-of-service on legitimate traffic also carrying the string. Thus, a clear area of research for us is to develop efficient mechanisms for comparing signatures with existing traffic corpus' – to understand the *impact* of filtering such traffic before we do so. However, even this approach may fall short against a sophisticated attacker with prior knowledge of an unreleased document. In this scenario an attacker might coerce Earlybird into blocking the documents release by simulating a worm containing substrings unique only to the unreleased document.

7.5 Coordination

One of the key benefits of signature extraction is that a given signature can be shared. This provides a "network effect" because the more deployments are made of a system such as ours, the more value there is to all deployments because of sharing. This sharing in turn can reduce response times, since the first site to discover a new worm signature can share it immediately. A more aggressive possibility is to add this detection capability to core routers which can then spread the signatures to edge networks. The issue of coordination brings up substantial questions related to trust, validation and policy that will require additional research attention to address.

8 Conclusions

New worm outbreaks routinely compromise hundreds of thousands of hosts and despite the enormous recovery costs incurred for past worms, we have been extremely fortunate in the degree of restraint demonstrated by worm authors. Thus the need for an adequate defense against future worm episodes is self-evident.

In this paper, we have described an approach for real-time detection of unknown worms and automated extraction of unique content signatures. Our content sifting algorithm efficiently analyses network traffic for prevalent and widely dispersed content strings – behavioral cues of worm activity. We have demonstrated that content sifting can be implemented with moderate memory and computational requirements and our untuned software-based prototype has been able to process over 200Mbps of live traffic. While the security field is inherently an "arms race", we believe that systems based on content sifting significantly raise the bar for worm authors. To wit, in our experience Earlybird has been able to detect and extract signatures for *all* contemporary worms and has also demonstrated that it can extract signatures for new, previously unknown, worms.

While we believe that EarlyBird can be a useful system in itself, we believe that the underlying method (maintaining state keyed by content signatures) may generalize to address a number of other interesting research problems. For example, we have found that slight modifications to Earlybird are able to detect large amounts of unsolicited bulk e-mail (SPAM) based on the same general principles as worm detection. Similarly, mass-intrusion attempts can also be revealed by this approach, as can denial-of-service attacks and peer-to-peer system activity.

Finally, the EarlyBird system demonstrates the feasibility of sophisticated wire-speed network security. While many industrial systems have only recently announced signature *detection* at Gigabit speeds, our experience with Earlybird suggests that signature *learning* at Gigabit speeds is equally viable. This leads us to hope

that other components of network security may also permit wire-speed implementation and allow security functions to be integrated – as a standard part of routers and switches – into the very heart of the network.

9 Acknowledgements

We would like to thank David Moore, Ramana Kompella and Geoff Voelker for their insightful discussions and Colleen Shannon, Jim Madden, Pat Wilson and David Visick for helping us understand the UCSD network. Finally, we would like to thank both the anonymous reviewers and our shepherd, David Wagner, for their constructive comments and suggestions. Support for this work was provided by NIST Grant 60NANB1D0118 and NSF Grant 0137102.

References

[1] Snort: Open source network intrusion detection system. www.snort.org, 2002.

[2] Carey Nachenberg. Method to analyze a program for presence of computer viruses by examining the opcode for faults before emulating instruction in emulator. U.S. Patent 5,964,889, Oct. 1999.

[3] Cisco Systems, Inc. Cisco Security Agent ROI: Deploying Intrusion Protection Agents on the Endpoint. Cisco Technical Whitepaper.

[4] F. Cohen. Computer Viruses — Theory and Experiments. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, Sept. 1984.

[5] F. Cohen. Computer Viruses — Theory and Experiments. *Computers and Security*, 6:22–35, 1987.

[6] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security and Privacy*, 2(4), July 2004.

[7] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2000.

[8] M. Erbschloe. Computer Economics VP Research Statement to Reuters News Service, Nov. 2001.

[9] C. Estan, S. Savage, and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2003.

[10] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2002.

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

[12] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, Oct. 1985.

[13] P. B. Gibbons and Y. Matias. New Sampling-based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of the ACM SIGMOD Conference*, June 1998.

[14] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.

[15] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th International Virus Bulletin Conference*, Sept. 1994.

[16] K.-A. Kim and B. Karp. Autograph: Toward Automated Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[17] C. Kreibich and J. Crowcroft. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the USENIX/ACM Workshop on Hot Topics in Networking*, Nov. 2003.

[18] J. Levin, R. LaBella, H. Owen, D. Contis, and B. Culver. The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*, June 2003.

[19] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, Sept. 2003.

[20] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*, 17–21 1994.

[21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. *IEEE Security and Privacy*, 1(4), July 2003.

[22] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the ACM Internet Measurement Workshop*, Nov. 2002.

[23] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *IEEE Proceedings of the INFOCOM*, Apr. 2003.

[24] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network Telescopes. Technical Report CS2004-0795, CSE Department, UCSD, July 2004.

[25] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.

[26] C. Morrow. BlackHole Route Server and Tracking Traffic on an IP Network. http://www.secsup.org/Tracking/.

[27] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the ACM SOSP Conference*, Oct. 2001.

[28] Network Associates Inc. McAfee Entercept Standard Edition. Product Datasheet.

[29] V. Paxson. Bro: a System for Detecting Network Intruders in Real-time. In *Proceedings of the USENIX Security Symposium*, Jan. 1998.

[30] T. H. Ptacek and T. N. Newsham. Insertion, Evasion and Denial-of-Service: Eluding Network Intrusion Detection. Technical report, Secure Networks Inc., Jan. 1998.

[31] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, Oct. 2003.

[32] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report 15-81, Center for Research in Computing Technology, Harvard University, 1981.

[33] J. Rochlis and M. Eichin. With Microscope and Tweezers: The Worm from MIT's Perspective. *Communications of the ACM*, 32(6):689–698, June 1989.

[34] S. Singh, C. Estan, G. Varghese, and S. Savage. Real-time Detection of Known and Unknown Worms. Technical Report CS2003-0745, CSE Department, UCSD, May 2003.

[35] S. Singh, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Real-time Detection of Unknown Worms. Technical Report CS2003-0761, CSE Department, UCSD, Aug. 2003.

[36] S. Singh, C. Estan, G. Varghese, and S. Savage. Accuracy bounds for the scaled bitmap data structure. Technical Report CS2004-0814, CSE Department, UCSD, Dec. 2004.

[37] S. Singh, G. Varghese, C. Estan, and S. Savage. Detecting Public Network Attacks using Signatures and Fast Content Analysis. United States Patent Application.

[38] A. C. Snoeren, C. Partridge, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP Traceback. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2001.

[39] E. Spafford. The Internet Worm: Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[40] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the ACM SIGOMM Conference*, Aug. 2000.

[41] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *to appear in the Journal of Computer Security*, 2004.

[42] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the USENIX Security Symposium*, Aug. 2002.

[43] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the USENIX Security Symposium*, Aug. 2003.

[44] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2004.

[45] N. Weaver. Personal communication, 2002.

[46] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.