Autograph: Toward Automated, Distributed Worm Signature Detection

Hyang-Ah Kim
hakim@cs.cmu.edu
Carnegie Mellon University

Brad Karp
brad.n.karp@intel.com, bkarp@cs.cmu.edu
Intel Research / Carnegie Mellon University

Abstract

Today's Internet intrusion detection systems (IDSes) monitor edge networks' DMZs to identify and/or filter malicious flows. While an IDS helps protect the hosts on its local edge network from compromise and denial of service, it cannot alone effectively intervene to halt and reverse the spreading of novel Internet worms. Generation of the *worm signatures* required by an IDS—the byte patterns sought in monitored traffic to identify worms—today entails non-trivial human labor, and thus significant delay: as network operators detect anomalous behavior, they communicate with one another and manually study packet traces to produce a worm signature. Yet intervention must occur early in an epidemic to halt a worm's spread. In this paper, we describe Autograph, a system that *automatically* generates signatures for novel Internet worms that propagate using TCP transport. Autograph generates signatures by analyzing the *prevalence of portions of flow payloads*, and thus uses no knowledge of protocol semantics above the TCP level. It is designed to produce signatures that exhibit high *sensitivity* (high true positives) and high *specificity* (low false positives); our evaluation of the system on real DMZ traces validates that it achieves these goals. We extend Autograph to share port scan reports among distributed monitor instances, and using trace-driven simulation, demonstrate the value of this technique in speeding the generation of signatures for novel worms. Our results elucidate the fundamental trade-off between early generation of signatures for novel worms and the specificity of these generated signatures.

1 Introduction and Motivation

In recent years, a series of Internet *worms* has exploited the confluence of the relative lack of diversity in system and server software run by Internet-attached hosts, and the ease with which these hosts can communicate. A worm program is self-replicating: it remotely exploits a software vulnerability on a victim host, such that the victim becomes infected, and itself begins remotely infecting other victims. The severity of the worm threat goes far beyond mere inconvenience. The total cost of the Code Red worm epidemic, as measured in lost productivity owing to interruptions in computer and network services, is estimated at $2.6 billion [7].

Motivated in large part by the costs of Internet worm epidemics, the research community has investigated worm propagation and how to thwart it. Initial investigations focused on case studies of the spreading of successful worms [8], and on comparatively modeling diverse propagation strategies future worms might use [18, 21]. More recently, researchers' attention has turned to methods for *containing* the spread of a worm. Broadly speaking, three chief strategies exist for containing worms by blocking their connections to potential victims: discovering ports on which worms appear to be spreading, and filtering all traffic destined for those ports; discovering source addresses of infected hosts and filtering all traffic (or perhaps traffic destined for a few ports) from those source addresses; and discovering the payload content string that a worm uses in its infection attempts, and filtering all flows whose payloads contain that content string.

Detecting that a worm appears to be active on a particular port [22] is a useful first step toward containment, but is often too blunt an instrument to be used alone; simply blocking all traffic for port 80 at edge networks across the Internet shuts down the entire web when a worm that targets web servers is released. Moore *et al.* [9] compared the relative efficacy of source-address filtering and content-based filtering. Their results show that content-based filtering of infection attempts slows the spreading of a worm more effectively: to confine an epidemic within a particular target fraction of the vulnerable host population, one may begin content-based filtering far later after the release of a worm than address-based filtering. Motivated by the efficacy of content-based filtering, we seek in this paper to answer the complementary question unanswered in prior work: *how should one obtain worm content signatures for use in content-based filtering?*

Here, a *signature* is a tuple `(IP-proto, dst-port, byteseq)`, where `IP-proto` is an IP protocol number, `dst-port` is a destination port number for that protocol, and `byteseq` is a variable-length, fixed sequence of bytes.[1] Content-based filtering consists of matching network flows (possibly requiring flow reassembly) against signatures; a match occurs when `byteseq` is found within the payload of a flow using the `IP-proto` protocol destined for `dst-port`. We restrict our investigation to worms that propagate over TCP in this work, and thus hereafter consider signatures as `(dst-port, byteseq)` tuples.

Today, there exist TCP-flow-matching systems that are "consumers" of these sorts of signatures. Intrusion detection systems (IDSes), such as Bro [11] and Snort [19], monitor all incoming traffic at an edge network's DMZ, perform TCP flow reassembly, and search for known worm signatures. These systems log the occurrence of inbound worm connections they observe, and can be configured (in the case of Bro) to change access control lists in the edge network's router(s) to block traffic from source IP addresses that have sent known worm payloads. Cisco's NBAR system [3] for routers searches for signatures in flow payloads, and blocks flows on the fly whose payloads are found to contain known worm signatures. We limit the scope of our inquiry to the *detection and generation* of signatures for use by these and future content-based filtering systems.

It is important to note that all the content-based filtering systems use databases of worm signatures that are *manually* generated: as network operators detect anomalous behavior, they communicate with one another, manually study packet traces to produce a worm signature, and publish that signature so that it may be added to IDS systems' signature databases. This labor-intensive, human-mediated process of signature generation is slow (on the order of hours or longer), and renders today's IDSes unhelpful in stemming worm epidemics—by the time a signature has been found manually by network operators, a worm may already have compromised a significant fraction of vulnerable hosts on the Internet.

We seek to build a system that automatically, without foreknowledge of a worm's payload or time of introduction, detects the signature of any worm that propagates by randomly scanning IP addresses. We assume the system monitors all inbound network traffic at an edge network's DMZ. *Autograph,* our worm signature detection system, has been designed to meet that goal. The system consists of three interconnected modules: a flow classifier, a content-based signature generator, and *tattler,* a protocol through which multiple distributed Autograph monitors may share information, in the interest of speeding detection of a signature that matches a newly released worm.

In our evaluation of Autograph, we explore two important themes. First, there is a trade-off between early detection of worm signatures and avoiding generation of signatures that cause false positives. Intuitively, early in an epidemic, worm traffic is less of an outlier against the background of innocuous traffic. Thus, targeting early detection of worm signatures increases the risk of mistaking innocuous traffic for worm traffic, and producing signatures that incur false positives. Second, we demonstrate the utility of distributed, collaborative monitoring in speeding detection of a novel worm's signature after its release.

In the remainder of this paper, we proceed as follows: In the next section, we catalog the goals that drove Autograph's design. In Section 3, we describe the detailed workings of a single Autograph monitor: its traffic classifier and content-based signature generator. Next, in Section 4, we evaluate the quality of the signatures Autograph finds when run on real DMZ traces from two edge networks. In Section 5 we describe tattler and the distributed version of Autograph, and using DMZ-trace-driven simulation evaluate the speed at which the distributed Autograph can detect signatures for newly introduced worms. After cataloging limitations of Autograph and possible attacks against it in Section 6, and describing related work in Section 7, we conclude in Section 8.

	high true +	low true +
high false +	sensitive, unspecific	insensitive, unspecific
low false +	sensitive, specific	insensitive, specific

Figure 1: Combinations of sensitivity and specificity.

2 Desiderata for a Worm Signature Detection System

Signature quality. Ideally, a signature detection system should generate signatures that match worms and only worms. In describing the efficacy of worm signatures in filtering traffic, we adopt the parlance used in epidemiology to evaluate a diagnostic test:

- *Sensitivity* relates to the *true positives* generated by a signature; in a mixed population of worm and non-worm flows, the fraction of the worm flows matched, and thus successfully identified, by the signature. Sensitivity is typically reported as $t \in [0,1]$, the fraction of true positives among worm flows.

- *Specificity* relates to the *false positives* generated by a signature; again, in a mixed population, the fraction of non-worm flows matched by the signature, and thus incorrectly identified as worms. Specificity is typically reported as $(1-f) \in [0,1]$, where f is the fraction of false positives among non-worm flows.

Throughout this paper, we classify signatures according to this terminology, as shown in Figure 1.

In practice, there is a tension between perfect sensitivity and perfect specificity; one often suffers when the other improves, because a diagnostic test (*e.g.,* "is this flow a worm or not?") typically measures only a narrow set of features in its input, and thus does not perfectly classify it. There may be cases where two inputs present with identical features in the eyes of a test, but belong in different classes. We examine this sensitivity-specificity trade-off in detail in Section 4.

Signature quantity and length. Systems that match flow payloads against signatures must compare a flow to all signatures known for its IP protocol and port. Thus, fewer signatures speed matching. Similarly, the cost of signature matching is proportional to the length of the signature, so short signatures may be preferable to long ones. Signature length profoundly affects specificity: when one signature is a subsequence of another, the longer one is expected to match fewer flows than the shorter one.

Robustness against polymorphic worms. A *polymorphic* worm[2] changes its payload in successive infection attempts. Such worms pose a particular challenge to match with signatures, as a signature sensitive to a portion of one worm payload may not be sensitive to any part of another worm payload. If a worm were "ideally" polymorphic, each of its payloads would contain no byte sequence in common with any other. That ideal is impossible, of course; single-byte sequences are shared by all payloads. In practice, a "strongly" polymorphic worm is one whose successive payloads share only very short byte subsequences in common. Such short subsequences, *e.g.,* 4 bytes long, cannot safely be used as worm signatures, as they may be insufficiently specific. Polymorphism generally causes an explosion in the number of signatures required to match a worm. An evaluation of the extent to which such worm payloads are achievable is beyond the scope of this paper. We note, however, that if a worm exhibits polymorphism, but does not change one or more relatively long subsequences across its variants, an efficient signature detection system will generate signatures that match these invariant subsequences, and thus minimize the number of signatures required to match all the worm's variants.

Timeliness of detection. Left unchecked by patches, traffic filtering, or other means, port-scanning worms infect vulnerable hosts at an exponential rate, until the infected population saturates. Provos [12] shows in simulation that patching of infected hosts is more effective the earlier it is begun after the initial release of a new worm, and that in practical deployment scenarios, patching must begin quickly (before 5% of vulnerable hosts become infected) in order to have hope of stemming an epidemic such that no more than 50% of vulnerable hosts ever become infected. Moore *et al.* [9] show similarly that signature-based filtering of worm traffic stops worm propagation most effectively when begun early.

Automation. A signature detection system should require minimal real-time operator intervention. Vetting signatures for specificity with human eyes, *e.g.,* is at odds with timeliness of signature detection for novel worms.

Application neutrality. Knowledge of application protocol semantics above the TCP layer (*e.g.,* HTTP, NFS RPCs, *&c.*) may be useful in distinguishing worm and innocuous traffic, and thus in producing signatures that are sensitive and specific. Avoiding leaning on such application-protocol knowledge, however, broadens the applicability of the signature detection system to all protocols layered atop TCP.

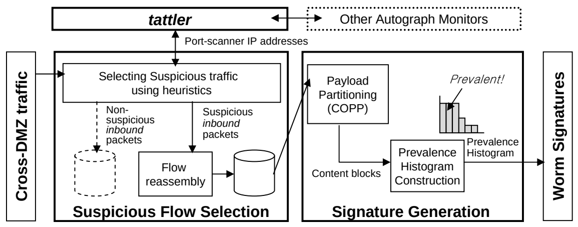
tattler
Other Autograph Monitors
Port-scanner IP addresses
Cross-DMZ traffic
Selecting Suspicious traffic using heuristics
Non-suspicious inbound packets
Suspicious inbound packets
Flow reassembly
Suspicious Flow Selection
Payload Partitioning (COPP)
Prevalent!
Content blocks
Prevalence Histogram Construction
Prevalence Histogram
Signature Generation
Worm Signatures

Figure 2: Architecture of an Autograph Monitor

Bandwidth efficiency. If a signature detection system is deployed in distributed fashion, such that traffic monitors communicate with one another about their observations, that communication should remain scalable, even when a worm generates tremendous network activity as it tries to spread. That is, monitor-to-monitor communication should grow slowly as worm activity increases.

3 Autograph System Design

Motivated by the design goals given in the previous section, we now present Autograph. We begin with a schematic overview of the system, shown in Figure 2. A single Autograph monitor's input is all traffic crossing an edge network's DMZ, and its output is a list of worm signatures. We defer discussion of tattler, used in distributed deployments of Autograph, to Section 5.2. There are two main stages in a single Autograph monitor's analysis of traffic. First, a *suspicious flow selection* stage uses heuristics to classify inbound TCP flows as either suspicious or non-suspicious.

After classification, packets for these inbound flows are stored on disk in a *suspicious flow pool* and *non-suspicious flow pool*, respectively. For clarity, throughout this paper, we refer to the output of the classifier using those terms, and refer to the *true* nature of a flow as *malicious* or *innocuous*. Further processing occurs *only* on payloads in the suspicious flow pool. Thus, flow classification reduces the volume of traffic that must be processed subsequently. We assume in our work that such heuristics will be far from perfectly accurate. Yet any heuristic that generates a suspicious flow pool in which truly malicious flows are a greater fraction of flows than in the total inbound traffic mix crossing the DMZ will likely reduce generation of signatures that cause false positives, by focusing Autograph's further processing on a flow population containing a lesser fraction of innocuous traffic. Autograph performs TCP flow reassembly for inbound payloads in the suspicious flow pool. The resulting reassembled payloads are analyzed in Autograph's second stage, *signature generation*.

We stress that Autograph segregates flows by destination port for signature generation; in the remainder of this paper, one should envision one separate instance of signature generation for each destination port, operating on flows in the suspicious flow pool destined for that port. Signature generation involves analysis of the *content* of payloads of suspicious flows to select sensitive and specific signatures. Two properties of worms suggest that content analysis may be fruitful. First, a worm propagates by exploiting one software vulnerability or a set of such vulnerabilities. That commonality in functionality has to date led to commonality in code, and thus in payload content, across worm infection payloads. In fact, Internet worms to date have had a single, unchanging payload in most cases. Even in those cases where multiple variants of a worm's payload have existed (*e.g.*, Nimda), those variants have shared significant overlapping content.[3] Second, a worm generates voluminous network traffic as it spreads; this trait stems from worms' self-propagating nature. For port-scanning worms, the exponential growth in the population of infected hosts and attendant exponential growth in infection attempt traffic are well known [8]. As also noted and exploited by Singh *et al.* [15], taken together, these two traits of worm traffic—content commonality and magnitude of traffic volume—suggest that analyzing the frequency of payload content should be useful in identifying worm payloads. During signature generation, Autograph measures the frequency with which non-overlapping payload substrings occur across all suspicious flow payloads, and proposes the most frequently occurring substrings as candidate signatures.

In the remainder of this section, we describe Autograph's two stages in further detail.

3.1 Selecting Suspicious Traffic

In this work, we use a simple port-scanner detection technique as a heuristic to identify malicious traffic; we classify all flows from port-scanning sources as suspicious. Note that we do not focus on the design of suspicious flow classifiers herein; Autograph can adopt *any* anomaly detection technique that classifies worm flows as suspicious with high probability. In fact, we deliberately use a port-scanning flow classifier because it is simple, computationally efficient, and *clearly imperfect*; our aim is to demonstrate that Autograph generates highly selective and specific signatures, even with a naive flow classifier. With more accurate flow classifiers, one will only expect the quality of Autograph's signatures to improve.

Many recent worms rely on scanning of the IP address space to search for vulnerable hosts while spreading. If a worm finds another machine that runs the desired service on the target port, it sends its infectious payload. Probing a non-existent host or service, however, results in an unsuccessful connection attempt, easily detectable by monitoring outbound ICMP host/port unreachable messages, or identifying unanswered inbound SYN packets. Hit-list worms [18], while not yet observed in the wild, violate this port-scanning assumption; we do not address them in this paper, but comment on them briefly in Section 6.

Autograph stores the source and destination addresses of each inbound unsuccessful TCP connection it observes. Once an external host has made unsuccessful connection attempts to more than s internal IP addresses, the flow classifier considers it to be a scanner. All successful connections from an IP address flagged as a scanner are classified as suspicious, and their inbound packets written to the suspicious flow pool, until that IP address is removed after a timeout (24 hours in the current prototype).[4] Packets held in the suspicious flow pool are dropped from storage after a configurable interval t. Thus, the suspicious flow pool contains all packets received from suspicious sources in the past time period t.[5]

Autograph reassembles all TCP flows in the suspicious flow pool. Every r minutes, Autograph considers initiating signature generation. It does so when for a single destination port, the suspicious flow pool contains more than a threshold number of flows θ. In an online deployment of Autograph, we envision typical r values on the order of ten minutes. We continue with a detailed description of signature generation in the next subsection.

3.2 Content-Based Signature Generation

Autograph next selects the most frequently occurring byte sequences across the flows in the suspicious flow pool as signatures. To do so, it divides each suspicious flow into smaller content blocks, and counts the number of suspicious flows in which each content block occurs. We term this count a content block's *prevalence*, and rank content blocks from most to least prevalent. As previously described, the intuition behind this ranking is that a worm's payload appears increasingly frequently as that worm spreads. When all worm flows contain a common, worm-specific byte sequence, that byte sequence will be observed in many suspicious flows, and so will be highly ranked.

Let us first describe how Autograph divides suspicious flows' payloads into shorter blocks. One might naively divide payloads into fixed-size, non-overlapping blocks, and compute the prevalence of those blocks across all suspicious flows. That approach, however, is brittle if worms even trivially obfuscate their payloads by reordering them, or inserting or deleting a few bytes. To see why, consider what occurs when a single byte is deleted or inserted from a worm's payload; all fixed-size blocks beyond the insertion or deletion will most likely change in content. Thus, a worm author could evade accurate counting of its substrings by trivial changes in its payload, if fixed-size, non-overlapping blocks were used to partition payloads for counting substring prevalence.

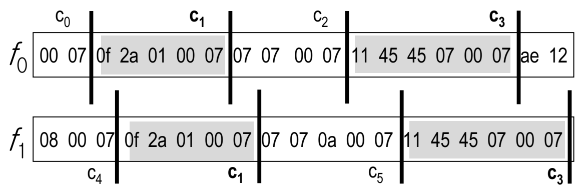
c0
c1
c2
c3
f0
00 07 0f 2a 01 00 07 07 07 00 07 11 45 45 07 00 07 ae 12
f1
08 00 07 0f 2a 01 00 07 07 07 0a 00 07 11 45 45 07 00 07
c4
c1
c5
c3

Figure 3: COPP with a breakmark of r(“0007”)

Instead, as first done in the file system domain in LBFS [10], we divide a flow's payload into *variable-length* content blocks using COntent-based Payload Partitioning (COPP). Because COPP determines the boundaries of each block based on payload content, the set of blocks COPP generates changes little under byte insertion or deletion.

To partition a flow's payload into content blocks, COPP computes a series of Rabin fingerprints r_i over a sliding k-byte window of the flow's payload, beginning with the first k bytes in the payload, and sliding one byte at a time toward the end of the payload. It is efficient to compute a Rabin fingerprint over a sliding window [13]. As COPP slides its window along the payload, it ends a content block when r_i matches a predetermined *breakmark*, B; when $r_i \equiv B \pmod{a}$.[6] The average content block size produced by COPP, a, is configurable; assuming random payload content, the window at any byte position within the payload equals the breakmark $B \pmod{a}$ with probability $1/a$.

Figure 3 presents an example of COPP, using a 2-byte window, for two flows f_0 and f_1. Sliding a 2-byte window from the first 2 bytes to the last byte, COPP ends a content block c_i whenever it sees the breakmark equal to the Rabin fingerprint for the byte string “0007”. Even if there exist byte insertions, deletions, or replacements between the two flows, COPP finds identical c_1 and c_3 blocks in both of them.

Because COPP decides content block boundaries probabilistically, there may be cases where COPP generates very short content blocks, or takes an entire flow's payload as a single content block. Very short content blocks are highly unspecific; they will generate many false positives. Taking the whole payload is not desirable either, because long signatures are not robust in matching worms that might vary their payloads. Thus, we impose minimum and maximum content block sizes, m and M, respectively. When COPP reaches the end of a content block and fewer than m bytes remain in the flow thereafter, it generates a content block that contains the last m bytes of the flow's payload. In this way, COPP avoids generating too short a content block, and avoids ignoring the end of the payload.

After Autograph divides every flow in the suspicious flow pool into content blocks using COPP, it discards content blocks that appear only in flows that originate from a single source IP address from further consideration. We found early on when applying Autograph to DMZ traces that such content blocks typically correspond to misconfigured or otherwise malfunctioning sources that are *not malicious*; such content blocks typically occur in many innocuous flows, and thus often lead to signatures that cause false positives. Singh *et al.* [15] also had this insight—they consider flow endpoint address distributions when generating worm signatures.

Suppose there are N distinct flows in the suspicious flow pool. Each remaining content block matches some portion of these N flows. Autograph repeatedly selects content blocks as signatures, until the selected set of signatures matches a configurable fraction w of the flows in the suspicious flow pool. That is, Autograph selects a signature set that “covers” at least wN flows in the suspicious flow pool.

We now describe how Autograph greedily selects content blocks as signatures from the set of remaining content blocks. Initially the suspicious flow pool F contains all suspicious flows, and the set of content blocks C contains all content blocks produced by COPP that were found in flows originating from more than one source IP address. Autograph measures the prevalence of each content block—the number of suspicious flows in F in which each content block in C appears—and sorts the content blocks from greatest to least prevalence. The content block with the greatest prevalence is chosen as the next signature. It is removed from the set of remaining content blocks C, and the flows it matches are removed from the suspicious flow pool, F. This entire process then repeats; the prevalence of content blocks in C in flows in F is computed, the most prevalent content block becomes a signature, and so on, until wN flows in the original F have been covered. This greedy algorithm attempts to minimize the size of the set of signatures by choosing the most prevalent content block at each step.

We incorporate a *blacklisting* technique into signature generation. An administrator may configure Autograph with a blacklist of disallowed signatures, in an effort to prevent the system from generating signatures that will cause false positives. The blacklist is simply a set of strings. Any signature Autograph selects that is a substring of an entry in the blacklist is discarded; Autograph eliminates that content block from C without selecting it as a signature, and continues as usual. We envision that an administrator may run Autograph for an initial *training period*, and vet signatures with human eyes during that period. Signatures generated during this period that match common patterns in innocuous flows (*e.g.*, `GET /index.html HTTP/1.0`) can be added to the blacklist.

At the end of this process, Autograph reports the selected set of signatures. The current version of the system publishes signature byte patterns in Bro's signature format, for direct use in Bro. Table 1 summarizes the parameters that control Autograph's behavior.

Note that because the flow classifier heuristic is imperfect, innocuous flows will unavoidably be included in the signature generation process. We expect two chief consequences

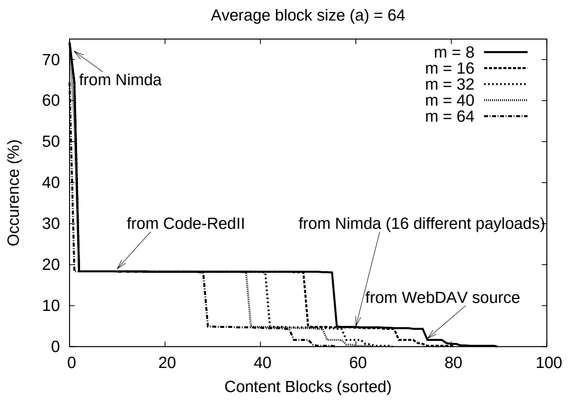

Figure 4: Prevalence histogram of content blocks, a=64 bytes, ICSI2 DMZ trace, day 3 (24 hrs).

of their inclusion:

Prevalent signatures matching innocuous and malicious flows. One possible result is that the probabilistic COPP process will produce content blocks that contain only protocol header or trailer data common to nearly *all* flows carrying that protocol, whether innocuous or malicious. Such blocks will top the prevalence histogram, but would clearly be abysmally unspecific if adopted for traffic filtering. To avoid choosing such unspecific content blocks, we can vary a and m toward longer block sizes.

Non-prevalent signatures for innocuous flows. Another possibility is that Autograph chooses a content block common to only a *few* innocuous flows. Such content blocks will not be prevalent, and will be at the tail of the prevalence histogram. Two heuristics can exclude these signatures from publication. First, by using a smaller w value, Autograph can avoid generation of signatures for the bottom $(1-w)$% of the prevalence distribution, though this choice may have the undesirable side effect of delaying detection of worms. The second useful heuristic comes from our experience with the initial COPP implementation. Figure 4 shows the prevalence histogram Autograph generates from a real DMZ trace. Among all content blocks, only a few are prevalent (those from Code-RedII, Nimda, and WebDAV) and the prevalence distribution has a noticeable tail. We can restrict Autograph to choose a content block as a signature only if more than p flows in the suspicious flow pool contain it, to avoid publishing signatures for non-prevalent content blocks.

4 Evaluation: Local Signature Detection

We now evaluate the quality of signatures Autograph generates. In this section, we answer the following two questions: First, how does content block size affect the the sensitivity and specificity of the signatures Autograph generates? And second, how robust is Autograph to worms that vary their payloads?

Symbol	Description
s	Port scanner detection threshold
a	COPP parameter: average content block size
m	COPP parameter: minimum content block size
M	COPP parameter: maximum content block size
w	Target percentage of suspicious flows to be represented in generated signatures
p	Minimum content block prevalence for use as signature
t	Duration suspicious flows held in suspicious flow pool
r	Interval between signature generation attempts
θ	Minimum size of suspicious flow pool to allow signature generation process

Table 1: Autograph's signature generation parameters.

Our experiments demonstrate that as content block size decreases, the likelihood that Autograph detects commonality across suspicious flows increases. As a result, as content block size decreases, Autograph generates progressively more sensitive but less specific signatures. They also reveal that small block sizes are more resilient to worms that vary their content, in that they can detect smaller common parts among worm payloads.

4.1 Offline Signature Detection on DMZ Traces

We first investigate the effect of content block size on the quality of the signatures generated by Autograph. In this subsection, we use a suspicious flow pool accumulated during an interval t of 24 hours, and consider only a single invocation of signature generation on that flow pool. No blacklisting is used in the results in this subsection, and filtering of content blocks that appear only from one source address before signature generation is disabled. All results we present herein are for a COPP Rabin fingerprint window of width $k = 4$ bytes.[7]

In our experiments, we feed Autograph one of three packet traces from the DMZs of two research labs; one from Intel Research Pittsburgh (Pittsburgh, USA) and two from ICSI (Berkeley, USA). IRP's Internet link was a T1 at the time our trace was taken, whereas ICSI's is over a 100 Mbps fiber to UC Berkeley. All three traces contain the full payloads of all packets. The ICSI and ICSI2 traces only contain inbound traffic to TCP port 80, and are IP-source-anonymized. Both sites have address spaces of 2^9 IP addresses, but the ICSI traces contain more port 80 traffic, as ICSI's web servers are more frequently visited than IRP's.

For comparison, we obtain the full list of HTTP worms in the traces using Bro with well-known signatures for the Code-Red, Code-RedII, and Nimda HTTP worms, and for an Agobot worm variant that exploits the WebDAV buffer overflow vulnerability (present only in the ICSI2 trace). Table 2 summarizes the characteristics of all three traces.

	IRP	ICSI	ICSI2
Measurement Period	Aug 1-7 2003 1 week	Jan 26 2004 24 hours	Mar 22-29 2004 1 week
Inbound HTTP packets	70K	793K	6353K
Inbound HTTP flows	26K	102K	825K
HTTP worm sources	72	351	1582
scanned	56	303	1344
not scanned	16	48	238
Nimda sources	18	57	254
CodeRed II sources	54	294	997
WebDav exploit sources	-	-	336
HTTP worm flows	375	1396	7127
Nimda flows	303	1022	5392
CodeRed flows	72	374	1365
WebDav exploit flows	-	-	370

Table 2: Summary of traces.

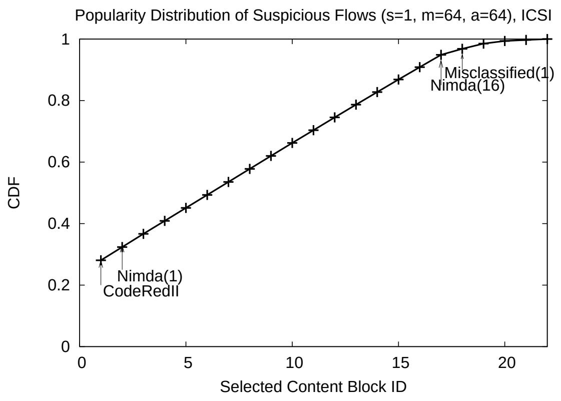

Figure 5: Prevalence of Selected Content Blocks in Suspicious Flow Pool, ICSI DMZ trace (24 hrs).

Autograph's suspicious flow classifier identifies unsuccessful connection attempts in each trace. For the IRP trace, Autograph uses ICMP host/port unreachable messages to compile the list of suspicious remote IP addresses. As neither ICSI trace includes outbound ICMP packets, Autograph infers failed connection attempts in those traces by looking at incoming TCP SYN and ACK pairs.

We run Autograph with varied scanner detection thresholds, $s \in \{1, 2, 4\}$. These thresholds are lower than those used by Bro and Snort, in the interest of catching as many worm payloads as possible (crucial early in an epidemic). As a result, our flow classifier misclassifies flows as suspicious more often, and more innocuous flows are submitted for signature generation.

We also vary the minimum content block size (m) and average content block size (a) parameters that govern COPP, but fix the maximum content block size (M) at 1024 bytes. We vary $w \in [10\%, 100\%]$ in our experiments. Recall that w limits the fraction of suspicious flows that may contribute content to the signature set. COPP adds content blocks to the signature set (most prevalent content block first, and then in order of decreasing prevalence) until one or more content blocks in the set match w percent of flows in the suspicious flow pool.

We first characterize the content block prevalence distribution found by Autograph with a simple example. Figure 5 shows the prevalence of content blocks found by COPP when we run COPP with $m = 64$, $a = 64$, and $w = 100\%$ over a suspicious flow pool captured from the full 24-hour ICSI trace with $s = 1$. At $w = 100\%$, COPP adds content blocks to the signature set until *all* suspicious flows are matched by one or more content blocks in the set. Here, the x axis represents the order in which COPP adds content blocks to the signature set (most prevalent first). The y axis represents the cumulative fraction of the population of suspicious flows containing any of the set of signatures, as the set of signatures grows. The trace contains Code-RedII, Nimda, and WebDAV worm flows. Nimda sources send 16 different flows with every infection attempt, to search for vulnerabilities under 16 different URLs. The first signature COPP generates matches Code-RedII; 28% of the suspicious flows are Code-RedII instances. Next, COPP selects 16 content blocks as signatures, one for each of the different payloads Nimda-infected machines transmit. About 5% of the suspicious flows are misclassified flows. We observe that commonality across those misclassified flows is insignificant. Thus, the content blocks from those misclassified flows tend to be lowly ranked.

To measure true positives (fraction of worm flows found), we run Bro with the standard set of policies to detect worms (distributed with the Bro software) on a trace, and then run Bro using the set of signatures generated by Autograph on that same trace. The true positive rate is the fraction of the total number of worms found by Bro's signatures (presumed to find all worms) also found by Autograph's signatures.

To measure false positives (fraction of non-worm flows matched by Autograph's signatures), we create a *sanitized* trace consisting of all non-worm traffic. To do so, we eliminate all flows from a trace that are identified by Bro as worms. We then run Bro using Autograph's signatures on the sanitized trace. The false positive rate is the fraction of all flows in the sanitized trace identified by Autograph's signatures as worms.

Because the number of false positives is very low compared to the total number of HTTP flows in the trace, we report our false positive results using the *efficiency* metric proposed by Staniford *et al.* [17]. Efficiency is the ratio of the number of true positives to the total number of positives, both false and true. Efficiency is proportional to the number of false positives, but shows the detail in the false positive trend when the false positive rate is low.

The graphs in Figure 6 show the sensitivity and the efficiency of the signatures generated by Autograph running on the full 24-hour ICSI trace for varied m. Here, we present experimental results for $s = 2$, but the results for other s are

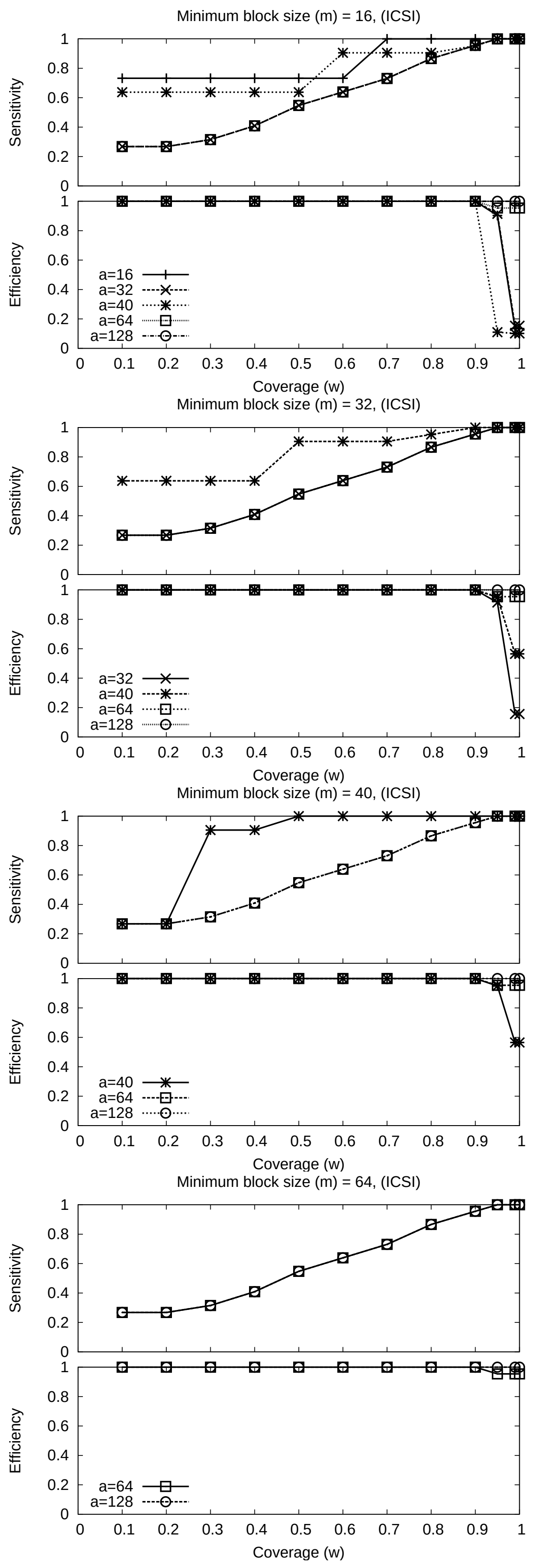

Figure 6: Sensitivity and Efficiency of Selected Signatures, ICSI DMZ trace (24 hrs).

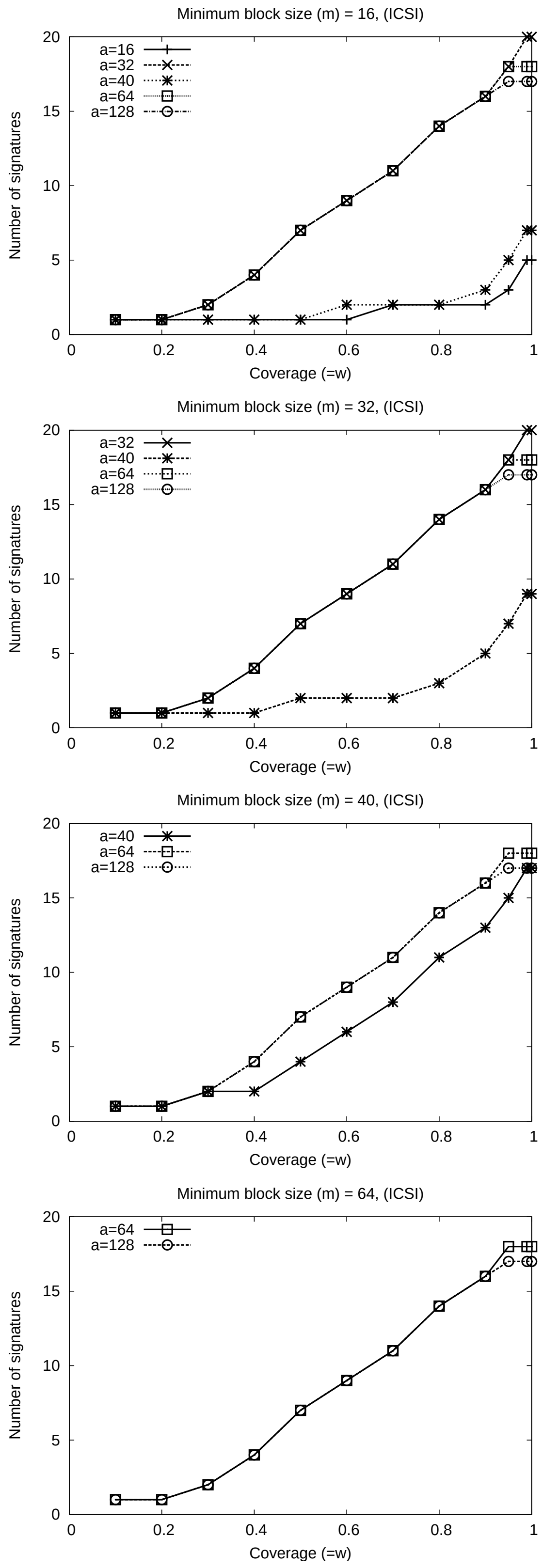

Figure 7: Number of Signatures, ICSI DMZ trace (24 hrs).

similar. Note that in these experiments, we apply the signatures Autograph generates from the 24-hour trace to the *same* 24-hour trace used to generate them.

The *x* axis varies *w*. As *w* increases, the set of signatures Autograph generates leads to greater sensitivity (fewer false negatives). This result is expected; greater *w* values cause Autograph to add content blocks to the signature set for an ever-greater fraction of the suspicious flow pool. Thus, if a worm appears rarely in the suspicious flow pool, and thus generates non-prevalent content blocks, those blocks will eventually be included in the signature set, for sufficiently large *w*.

However, recall from Figure 5 that about 5% of the suspicious flows are innocuous flows that are misclassified by the port-scanner heuristic as suspicious. As a result, for $w > 95\%$, COPP risks generating a less specific signature set, as COPP begins to select content blocks from the innocuous flows. Those content blocks are most often HTTP trailers, found in common across misclassified innocuous flows.

For this trace, COPP with $w \in [90\%, 94.8\%]$ produces a set of signatures that is *perfect:* it causes 0 false negatives and 0 false positives. Our claim is *not* that this *w* parameter value is valid for traces at different sites, or even at different times; on the contrary, we expect that the range in which no false positives and no false negatives occurs is sensitive to the details of the suspicious flow population. Note, however, that the existence of a range of *w* values for which perfect sensitivity and specificity are possible serves as a very preliminary validation of the COPP approach—if no such range existed for this trace, COPP would always be forced to trade false negatives for false positives, or vice-versa, for *any w* parameter setting. Further evaluation of COPP on a more diverse and numerous set of traffic traces is clearly required to determine whether such a range exists for a wider range of workloads.

During examination of the false positive cases found by Autograph-generated signatures when $w > 94.8\%$, we noted with interest that Autograph's signatures detected Nimda sources *not* detected by Bro's stock signatures. There are only three stock signatures used by Bro to spot a Nimda source, and the Nimda sources in the ICSI trace did not transmit those particular payloads. We removed these few cases from the count of false positives, as Autograph's signatures *correctly* identified them as worm flows, and thus we had *erroneously* flagged them as false positives by assuming that any flow not caught by Bro's stock signatures is not a worm.

We now turn to the effect of content block size on the specificity and the number of signatures Autograph generates. Even in the presence of innocuous flows misclassified as suspicious, the largest average and minimum content block sizes (such as 64 and 128 bytes) avoid most false positives; efficiency remains close to 1. We expect this result because increased block size lowers the probability of finding common content across misclassified flows during the signature generation process. Moreover, as signature length increases, the number of innocuous flows that match a signature decreases. Thus, choosing larger *a* and *m* values will help Autograph avoid generating signatures that cause false positives.

Note, however, there is a trade-off between content block length and the number of signatures Autograph generates, too. For large *a* and *m*, it is more difficult for COPP to detect commonality across worm flows unless the flows are identical. So as *a* and *m* increase, COPP must select more signatures to match any group of variants of a worm that contain some common content. The graphs in Figure 7 present the size of the signature set Autograph generates as a function of *w*. For smaller *a* and *m*, Autograph needs fewer content blocks to cover *w* percent of the suspicious flows. In this trace, for example, COPP can select a short byte sequence in common across different Nimda payload variants (*e.g.*, `cmd.exe?c+dir HTTP/1.0..Host:www..Connection: close....`) when we use small *a* and *m*, such as 16. The size of the signature set becomes a particular concern when worms aggressively vary their content across infection attempts, as we discuss in the next section. Before continuing on, we note that results obtained running Autograph on the IRP and ICSI2 traces are quite similar to those reported above, and are therefore elided in the interest of brevity.

4.2 Polymorphic and Metamorphic Worms

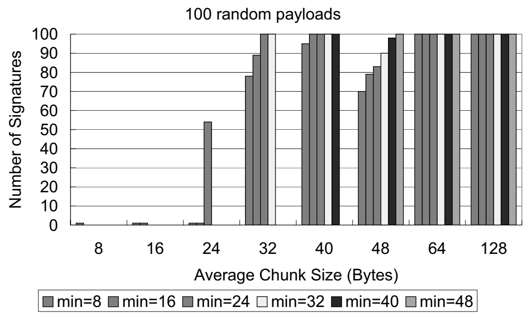

Figure 8: Content block size *vs.* number of signatures.

We expect short content blocks to be most robust against worms that vary their content, such as polymorphic worms, which encrypt their content differently on each connection, and metamorphic worms, which obfuscate their instruction sequences on each connection. Unfortunately (fortunately?) no such Internet worm has yet been reported in the wild. To test Autograph's robustness against these varying worms, we generate a synthetic polymorphic worm based on the Code-RedII payload. A Code-RedII worm payload consists of a regular HTTP GET header, more than 220 filler characters, a sequence of Unicode, and the main worm executable code. The Unicode sequence causes a buffer overflow and transfers execution flow to the subsequent worm binary. We use *random values* for all filler bytes, and even for the worm code, but leave the HTTP GET command and 56-byte Unicode sequence fixed. This degree of variation in content is more severe than that introduced by the various obfuscation techniques discussed by Christodorescu *et al.* [2]. As shown in Figure 8, when a relatively short, invariant string is present in a polymorphic or metamorphic worm, Autograph can find a short signature that matches it, when run with small average and minimum content block sizes. However, such short content block sizes may be unspecific, and thus yield signatures that cause false positives.

5 Evaluation: Distributed Signature Detection

Our evaluation of Autograph in the preceding section focused chiefly on the behavior of a single monitor's content-based approach to signature generation. That evaluation considered the case of offline signature detection on a DMZ trace 24 hours in length. We now turn to an examination of Autograph's speed in detecting a signature for a *new* worm after the worm's release, and demonstrate that operating multiple, distributed instances of Autograph significantly speeds this process, *vs.* running a single instance of Autograph on a single edge network. We use a combination of simulation of a worm's propagation and DMZ-trace-driven simulation to evaluate the system in the online setting; our sense of ethics restrains us from experimentally measuring Autograph's speed at detecting a novel worm *in vivo.*

Measuring how quickly Autograph detects and generates a signature for a newly released worm is important because it has been shown in the literature that successfully containing a worm requires early intervention. Recall that Provos' results [12] show that reversing an epidemic such that fewer than 50% of vulnerable hosts ever become infected can require intervening in the worm's propagation before 5% of vulnerable hosts are infected. Two delays contribute to the total delay of signature generation:

- How long must an Autograph monitor wait until it accumulates enough worm payloads to generate a signature for that worm?
- Once an Autograph monitor receives sufficient worm payloads, how long will it take to generate a signature for the worm, given the background "noise" (innocuous flows misclassified as suspicious) in the trace?

We proceed now to measure these two delays.

5.1 Single *vs.* Multiple Monitors

Let us now measure the time required for an Autograph monitor to accumulate worm payloads after a worm is released. We first describe our simulation methodology for simulating a Code-RedI-v2-like worm, which is after that of Moore *et al.* [9]. We simulate a vulnerable population of 338,652

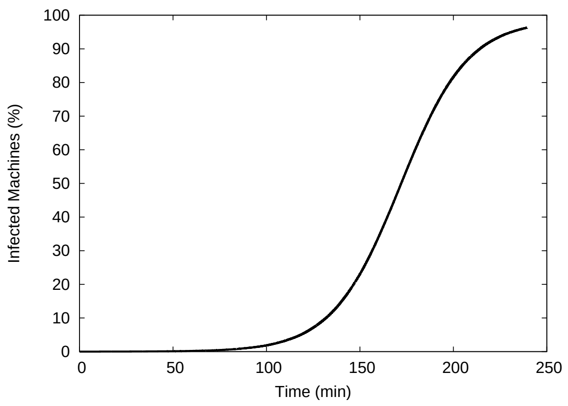

Figure 9: Infection progress for a simulated Code-RedI-v2-like worm.

hosts, the number of infected source IPs observed in [8] that are uniquely assignable to a single Autonomous System (AS) in the BGP table data (obtained from RouteViews [20]) of the 19th of July, 2001, the date of the Code-Red outbreak. There are 6378 ASes that contain at least one such vulnerable host in the simulation. Unlike Moore *et al.*, we do not simulate the reachability among ASes in that BGP table; we make the simplifying assumption that all ASes may reach all other ASes. This assumption may cause the worm to spread somewhat faster in our simulation than in Moore *et al.*'s. We assign actual IP address ranges for real ASes from the BGP table snapshot to each AS in the simulation, according to a truncated distribution of the per-AS IP address space sizes from the entire BGP table snapshot. The distribution of address ranges we assign is truncated in that we avoid assigning any address blocks larger than /16s to any AS in the simulation. We avoid large address blocks for two reasons: first, few such monitoring points exist, so it may be unreasonable to assume that Autograph will be deployed at one, and second, a worm programmer may trivially code a worm to avoid scanning addresses within a /8 known to harbor an Autograph monitor. Our avoidance of large address blocks only lengthens the time it will take Autograph to generate a worm signature after a novel worm's release. We assume 50% of the address space within the vulnerable ASes is populated with reachable hosts, that 25% of these reachable hosts run web servers, and we fix the 338,652 vulnerable web servers uniformly at random among the total population of web servers in the simulation. Finally, the simulated worm propagates using random IP address scanning over the entire 2^{28} non-class-D IP address space, and a probe rate of 10 probes per second. We simulate network and processing delays, randomly chosen in $[0.5, 1.5]$ seconds, between a victim's receipt of an infecting connection and its initiation of outgoing infection attempts. We begin the epidemic by infecting 25 vulnerable hosts at time zero. Figure 9 shows the growth of the epidemic within the vulnerable host population over time.

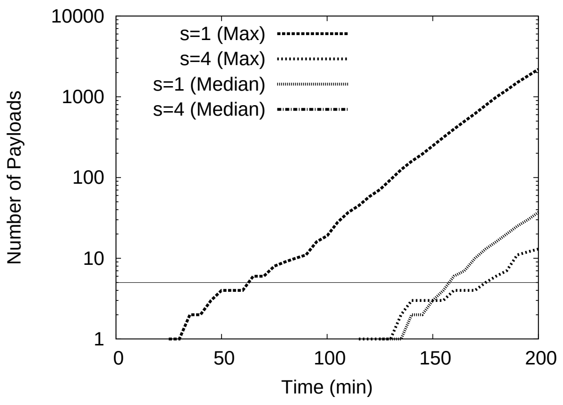

Figure 10: Payloads observed over time: single, isolated monitors.

In these first simulations, we place Autograph monitors at a randomly selected 1% of the ASes that include vulnerable hosts (63 monitors). Figure 10 shows the maximum and median numbers of payloads detected over time across all monitors; note that the *y* axis is log-scaled. First, let us consider the case where only a single site on the Internet deploys Autograph on its network. In this case, it is the median time required by all 63 monitors to detect a given number of flows that approximates the expected time for a singleton monitor to do the same. When monitors identify port scanners aggressively, after a single failed connection from a source address ($s = 1$), the median monitor accumulates 5 worm payloads after over 9000 seconds. Using the more conservative port-scan threshold $s = 4$, the median monitor accumulates *no* payloads within 10000 seconds. These results are not encouraging—from Figure 9, we know that after 9000 seconds (150 minutes), over 25% of vulnerable hosts have been infected.

Now let us consider the case where 63 monitors are all in active use simultaneously and distributedly. If we presume that the first monitor to generate a signature for the worm may (nearly) instantly disseminate that signature to all who wish to filter worm traffic, by application-level multicast [1] or other means, the earliest Autograph can possibly find the worm's signature is governed by the "luckiest" monitor in the system—the first one to accumulate the required number θ of worm payloads. The "luckiest" monitor in this simulated distributed deployment detects 5 worm payloads shortly before 4000 seconds have elapsed. This result is far more encouraging—after 4000 seconds (66 minutes), fewer than 1% of vulnerable hosts have been infected. Thus, provided that all Autograph monitors disseminate the worm signatures they detect in a timely fashion, there is immense benefit in the speed of detection of a signature for a novel worm when Autograph is deployed distributedly, even at as few as 1% of ASes that contain vulnerable hosts.

Using the more conservative port-scan threshold $s = 4$, the monitor in the distributed system to have accumulated the most worm payloads after 10000 seconds has still only collected 4. Here, again, we observe that targeting increased specificity (by identifying suspicious flows more conservatively) comes at a cost of reduced sensitivity; in this case, sensitivity may be seen as the number of worm flows matched *over time.*

Running multiple independent Autograph monitors clearly pays a dividend in faster worm signature detection. A natural question that follows is whether detection speed might be improved further if the Autograph monitors shared information with one another in some way.

5.2 tattler: Distributed Gathering of Suspect IP Addresses

At the start of a worm's propagation, the aggregate rate at which all infected hosts scan the IP address space is quite low. Because Autograph relies on overhearing unsuccessful scans to identify suspicious source IP addresses, early in an epidemic an Autograph monitor will be slow to accumulate suspicious addresses, and in turn slow to accumulate worm payloads. We now introduce an extension to Autograph named *tattler* that, as its name suggests, shares suspicious source addresses among all monitors, toward the goal of accelerating the accumulation of worm payloads.

We assume in the design of tattler that a multicast facility is available to all Autograph monitors, and that they all join a single multicast group. While IP multicast is not a broadly deployed service on today's Internet, there are many viable end-system-oriented multicast systems that could provide this functionality, such as Scribe [1]. In brief, Autograph monitor instances could form a Pastry overlay, and use Scribe to multicast to the set of all monitors. We further assume that users are willing to publish the IP addresses that have been port scanning them.[8]

The tattler protocol is essentially an application of the RTP Control Protocol (RTCP) [14], originally used to control multicast multimedia conferencing sessions, slightly extended for use in the Autograph context. The chief goal of RTCP is to allow a set of senders who all subscribe to the same multicast group to share a capped quantity of bandwidth fairly. In Autograph, we seek to allow monitors to announce to others the `(IP-addr, dst-port)` pairs they have observed port scanning themselves, to limit the total bandwidth of announcements sent to the multicast group within a predetermined cap, and to allocate announcement bandwidth relatively fairly among monitors. We recount the salient features of RTCP briefly:

- A population of senders all joins the same multicast group. Each is configured to respect the same total bandwidth limit, *B*, for the aggregate traffic sent to the group.
- Each sender maintains an interval value *I* it uses between its announcements. Transmissions are jittered uniformly

at random within $[0.5, 1.5]$ times this timer value.

- Each sender stores a list of the unique source IP addresses from which it has received announcement packets. By counting these, each sender learns an estimate of the total number of senders, N. Entries in the list expire if their sources are not heard from within a timeout interval.

- Each sender computes $I = N/B$. Senders keep a running average of the sizes of all announcement packets received, and scale I according to the size of the announcement they wish to send next.

- When too many senders join in a brief period, the aggregate sending rate may exceed C. RTCP uses a *reconsideration* procedure to combat this effect, whereby senders lengthen I probabilistically.

- Senders which depart may optionally send a BYE packet in compliance with the I inter-announcement interval, to speed other senders' learning of the decrease in the total group membership.

- RTCP has been shown to scale to thousands of senders.

In the tattler protocol, each announcement a monitor makes contains between one and 100 port-scanner reports of the form `(src-IP, dst-port)`. Monitors only announce scanners they've heard *themselves*. Hearing a report from another monitor for a scanner suppresses announcement of that scanner for a *refresh interval*. After a *timeout interval*, a monitor expires a scanner entry if that scanner has not directly scanned it and no other monitor has announced that scanner. Announcement packets are sent in accordance with RTCP. Every time the interval I expires, a monitor sends any announcements it has accumulated that haven't been suppressed by other monitors' announcements. If the monitor has no port scans to report, it instead sends a BYE, to relinquish its share of the total report channel bandwidth to other monitors.

Figure 11 shows the bandwidth consumed by the tattler protocol during a simulated Code-RedI-v2 epidemic, for three deployed monitor populations (6, 63, and 630 monitors). We use an aggregate bandwidth cap C of 512 Kbps in this simulation. Note that the peak bandwidth consumed across all deployments is a mere 15Kbps. Thus, sharing port scanner information among monitors is quite tractable. While we've not yet explicitly explored dissemination of signatures in our work thus far, we expect a similar protocol to tattler will be useful and scalable for advertising signatures, both to Autograph monitors and to other boxes that may wish to filter using Autograph-generated signatures.

Note well that "background" port scanning activities unrelated to the release of a new worm are prevalent on the Internet, and tattler must tolerate the load caused by

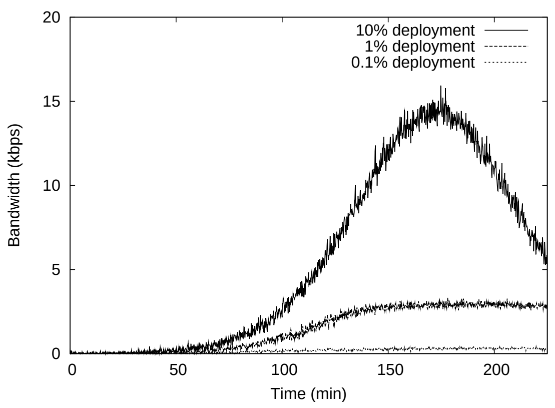

Figure 11: Bandwidth consumed by tattler during a Code-RedI v2 epidemic, for varying numbers of deployed monitors.

such background port scanning. `dshield.org` [4] reports daily measurements of port scanning activities, as measured by monitors that cover approximately 2^{19} IP addresses. The `dshield.org` statistics from December 2003 and January 2004 suggest that approximately 600,000 unique `(source-IP, dst-port)` pairs occur in a 24-hour period. If we conservatively double that figure, tattler would have to deliver 1.2M reports per day. A simple back-of-the-envelope calculation reveals that tattler would consume 570 bits/second to deliver that report volume, assuming one announcement packet per `(source-IP, dst-port)` pair. Thus, background port scanning as it exists in today's Internet represents insignificant load to tattler.

We now measure the effect of running tattler on the time required for Autograph to accumulate worm flow payloads in a distributed deployment. Figure 12 shows the time required to accumulate payloads in a deployment of 63 monitors that use tattler. Note that for a port scanner detection threshold $s = 1$, the shortest time required to accumulate 5 payloads across monitors has been reduced to approximately 1500 seconds, from nearly 4000 seconds without tattler (as shown in Figure 10). Thus, sharing scanner address information among monitors with tattler speeds worm signature detection.

In sum, running a distributed population of Autograph monitors holds promise for speeding worm signature detection in two ways: it allows the "luckiest" monitor that *first* accumulates sufficient worm payloads determine the delay until signature detection, and it allows monitors to chatter about port-scanning source addresses, and thus *all monitors* classify worm flows as suspicious earlier.

5.3 Online, Distributed, DMZ-Trace-Driven Evaluation

The simulation results presented thus far have quantified the time required for Autograph to accumulate worm payloads

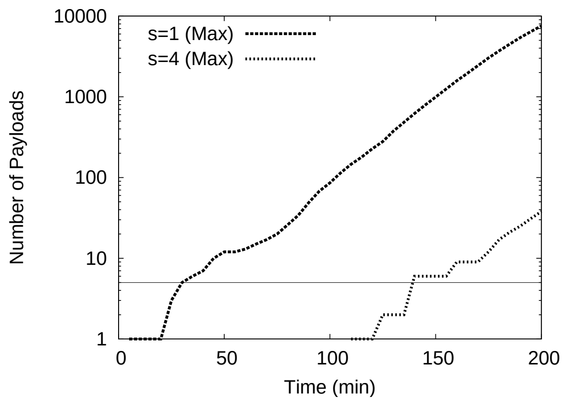

Figure 12: Payloads observed over time: tattler among distributed monitors.

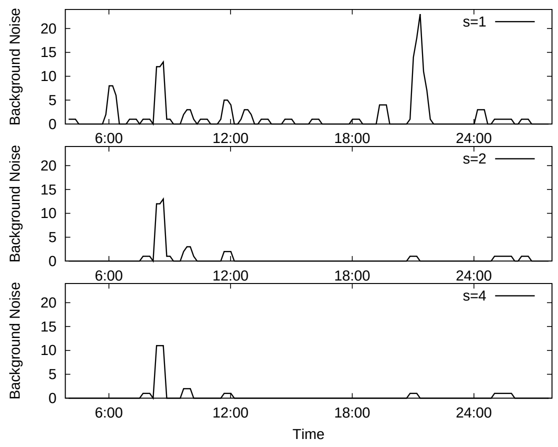

Figure 13: Background "noise" flows classified as suspicious *vs.* time, with varying port-scanner thresholds; ICSI DMZ trace.

after a worm's release. We now use DMZ-trace-driven simulation on the one-day ICSI trace to measure how long it takes Autograph to identify a newly released worm *among the background noise of flows that are not worms,* but have been categorized by the flow classifier as suspicious after port scanning the monitor. We are particularly interested in the trade-off between early signature generation (sensitivity across time, in a sense) and specificity of the generated signatures. We measure the speed of signature generation by the fraction of vulnerable hosts infected when Autograph first detects the worm's signature, and the specificity of the generated signatures by counting the *number* of signatures generated that cause false positives. We introduce this latter metric for specificity because raw specificity is difficult to interpret: if a signature based on non-worm-flow content (from a misclassified innocuous flow) is generated, the number of false positives it causes depends strongly on the traffic mix at that particular site. Furthermore, an unspecific signature may be relatively straightforward to identify as such with "signature blacklists" (disallowed signatures that should not be used for filtering traffic) provided by a system operator.[9]

We simulate an online deployment of Autograph as follows. We run a single Autograph monitor on the ICSI trace. To initialize the list of suspicious IP addresses known to the monitor, we run Bro on the *entire* 24-hour trace using all known worm signatures, and exclude worm flows from the trace. We then scan the *entire* resulting worm-free 24-hour trace for port scan activity, and record the list of port scanners detected with thresholds of $s \in \{1,2,4\}$. To emulate the steady-state operation of Autograph, we populate the monitor's suspicious IP address list with the *full* set of port scanners from one of these lists, so that all flows from these sources will be classified as suspicious. We can then generate a *background noise* trace, which consists of only non-worm flows from port scanners, as would be detected by a running Autograph monitor for each of $s \in \{1,2,4\}$. Figure 13 shows the quantity of non-worm noise flows in Autograph's suspicious traffic pool over the trace's full 24 hours.

We simulate the release of a novel worm at a time of our choosing within the 24-hour trace as follows. We configure Autograph with a signature generation periodicity r of 10 minutes, and a holding period t for the suspicious flow pool of 30 minutes. Using the simulation results from Section 5.2, we count the number of worm flows *expected* to have been accumulated by the "luckiest" monitor among the 63 deployed during each 30-minute period, at intervals of 10 minutes. We then add that number of complete Code-RedI-v2 flows (available from the pristine, unfiltered trace) to the suspicious traffic pool from the corresponding 30-minute portion of the ICSI trace, to produce a realistic mix of DMZ-trace noise and the expected volume of worm traffic (as predicted by the worm propagation simulation). In these simulations, we vary θ, the total number of flows that must be found in the suspicious traffic pool to cause signature generation to be triggered. All simulations use $w = 95\%$. Because the quantity of noise varies over time, we uniformly randomly choose the time of the worm's introduction, and take means over ten simulations.

Figure 14 shows the fraction of the vulnerable host population that is infected when Autograph detects the newly released worm as a function of θ, for varying port scanner detection sensitivities/specificities ($s \in \{1,2,4\}$). Note the log-scaling of the x axis. These results demonstrate that for a very sensitive/unspecific flow classifier ($s = 1$), across a wide range of θs (between 1 and 40), Autograph generates a signature for the worm before the worm spreads to even 1% of vulnerable hosts. As the flow classifier improves in specificity but becomes less sensitive ($s = \{2,4\}$), Autograph's generation of the worm's signature is delayed, as expected.

Figure 15 shows the number of unspecific (false-positive-inducing) signatures generated by Autograph, as a function of θ, for different sensitivities/specificities of flow classifier. The

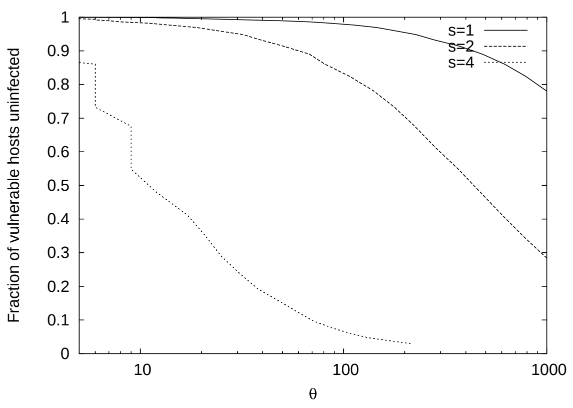

Figure 14: Fraction of vulnerable hosts uninfected when worm signature detected *vs.* θ, number of suspicious flows required to trigger signature detection.

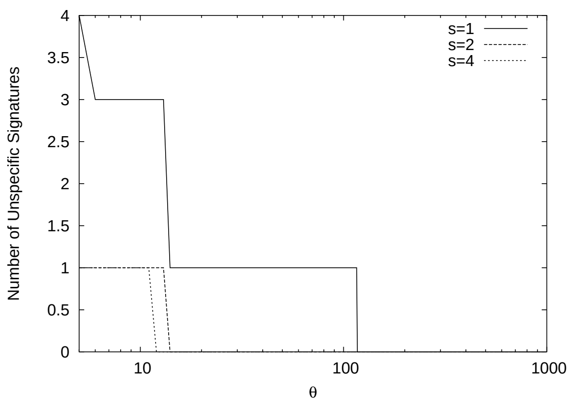

Figure 15: Number of unspecific signatures generated *vs.* θ, number of suspicious flows required to trigger signature detection.

goal, of course, is for the system to generate zero unspecific signatures, but to generate a worm signature before the worm spreads too far. Our results show that for $s = 2$ and $\theta = 15$, Autograph generates signatures that cause no false positives, yet generates the signature for the novel worm before 2% of vulnerable hosts become infected. Our point is *not* to argue for these particular parameter values, but rather to show that there exists a region of operation where the system meets our stated design goals. More importantly, though, these results show that an improved flow classifier improves Autograph—as flow classifiers benefit from further research and improve, Autograph can adopt these improvements to offer faster worm signature generation with lower false positive rates.

6 Attacks and Limitations

We briefly catalog a few attacks that one might mount against Autograph, and limitations of the current system.

Overload. Autograph reassembles suspicious TCP flows. Flow reassembly is costly in state in comparison with processing packets individually, but defeats the subterfuge of fragmenting a worm's payload across many small packets [11]. We note that the number of inbound flows a monitor observes may be large, in particular after a worm spreads successfully. If Autograph tries to reassemble every incoming suspicious flow, it may be susceptible to DoS attack. We note that Autograph treats all destination ports separately, and thus parallelizes well across ports; a site could run multiple instances of Autograph on separate hardware, and thus increase its aggregate processing power, for flow reassembly and all other processing. Autograph may also sample suspicious flows when the number of suspicious flows to process exceeds some threshold; we intend to investigate this heuristic in future.

Source-address-spoofed port scans. Port scans from spoofed IP source addresses are a peril for most IDSes. The chief reason for monitoring port scans is to limit the damage their originators can inflict, most often by filtering packets that originate from known port scanners. Such filtering invites attackers to spoof port scans from the IP addresses of those whose traffic they would like to block [11, 5]. Source-spoofed port scans can be used to mount different attacks, more specific to Autograph: the tattler mechanism must carry report traffic proportional to the number of port scanners. An attacker could attempt to saturate tattler's bandwidth limit with spoofed scanner source addresses, and thus render tattler useless in disseminating addresses of *true* port scanners. A source-spoofing attacker could also cause a remote source's traffic to be included by Autograph in signature generation.

Fortunately, a simple mechanism holds promise for rendering both these attacks ineffective. Autograph classifies an inbound SYN destined for an unpopulated IP address or port with no listening process as a port scan. To identify TCP port scans from spoofed IP source addresses, an Autograph monitor could respond to such inbound SYNs with a SYN/ACK, provided the router and/or firewall on the monitored network can be configured not to respond with an ICMP host or port unreachable. If the originator of the connection responds with an ACK with the appropriate sequence number, the source address on the SYN could not have been spoofed. The monitor may thus safely view all source addresses that send proper ACK responses to SYN/ACKs as port scanners. Non-ACK responses to these SYN/ACKs (RSTs or silence) can then be ignored; *i.e.*, the source address of the SYN is not recorded as a port scanner. Note that while a non-source-spoofing port scanner may *choose* not to respond with an ACK, any source that hopes to complete a connection and successfully transfer an infecting payload *must* respond with an ACK, and thus identify itself as a port scanner. Jung *et al.* independently propose this same technique in [5].

Hit-list scanning. If a worm propagates using a hit list [18], rather than by scanning IP addresses that may or may not correspond to listening servers, Autograph's port-scan-based suspicious flow classifier will fail utterly to include that worm's payloads in signature generation. Identifying worm flows that propagate by hit lists is beyond the scope of this paper. We are unaware at this writing of any published system that detects such flows; state-of-the-art malicious payload gathering methods, such as honeypots, are similarly stymied by hit-list propagation. Nevertheless, any future innovation in the detection of flows generated by hit-list-using worms may be incorporated into Autograph, to augment or replace the naive port-scan-based heuristic used in our prototype.

7 Related Work

Singh *et al.* [15] generate signatures for novel worms by measuring packet content prevalence and address dispersion at a single monitoring point. Their system, EarlyBird, avoids the computational cost of flow reassembly, but is susceptible to attacks that spread worm-specific byte patterns over a sequence of short packets. Autograph instead incurs the expense of flow reassembly, but mitigates that expense by *first* identifying suspicious flows, and *thereafter* performing flow reassembly and content analysis only on those flows. EarlyBird reverses these stages; it finds sub-packet content strings first, and applies techniques to filter out innocuous content strings second. Autograph and EarlyBird both make use of Rabin fingerprints, though in different ways: Autograph's COPP technique uses them as did LBFS, to break flow payloads into non-overlapping, variable-length chunks efficiently, based on payload content. EarlyBird uses them to generate hashes of overlapping, fixed-length chunks at every byte offset in a packet efficiently. Singh *et al.* independently describe using a white-list to disallow signatures that cause false positives (described herein as a blacklist for signatures, rather than a white-list for traffic), and report examples of false positives that are prevented with such a white-list [16].

Kreibich and Crowcroft [6] describe Honeycomb, a system that gathers suspicious traffic using a honeypot, and searches for least common substrings in that traffic to generate worm signatures. Honeycomb relies on the inherent suspiciousness of traffic received by a honeypot to limit the traffic considered for signature generation to truly suspicious flows. This approach to gathering suspicious traffic is complementary to that adopted in Autograph; we intend to investigate acquiring suspicious flows using honeypots for signature generation by Autograph in future. The evaluation of Honeycomb assumes all traffic received by a honeypot is suspicious; that assumption may not always hold, in particular if attackers deliberately submit innocuous traffic to the system. Autograph, Honeycomb, and EarlyBird will face that threat as knowledge of their deployment spreads; we believe vetting candidate signatures for false positives among many distributed monitors may help to combat it.

Provos [12] observes the complementary nature of honeypots and content-based signature generation; he suggests providing payloads gathered by `honeyd` to Honeycomb. We observe that Autograph would similarly benefit from `honeyd`'s captured payloads. Furthermore, if `honeyd` participated in tattler, Autograph's detection of suspicious IP addresses would be sped, with less communication than that required to transfer complete captured payloads from instances of `honeyd` to instances of Autograph.

Yegneswaran *et al.* [23] corroborate the benefit of distributed monitoring, both in speeding the accurate accumulation of port scanners' source IP addresses, and in speeding the accurate determination of port scanning volume. Their DOMINO system detects port scanners using active-sinks (honeypots), both to generate source IP address blacklists for use in address-based traffic filtering, and to detect an increase in port scanning activity on a port with high confidence. The evaluation of DOMINO focuses on speed and accuracy in determining port scan volume and port scanners' IP addresses, whereas our evaluation of Autograph focuses on speed and accuracy in generating worm signatures, as influenced by the speed and accuracy of worm payload accumulation.

Our work is the first we know to evaluate the tradeoff between earliness of detection of a novel worm and generation of signatures that cause false positives in content-based signature detection.

8 Conclusion and Future Work

In this paper, we present design criteria for an automated worm signature detection system, and the design and evaluation of Autograph, a DMZ monitoring system that is a first step toward realizing them. Autograph uses a naive, port-scan-based flow classifier to reduce the volume of traffic on which it performs content-prevalence analysis to generate signatures. The system ranks content according to its prevalence, and only generates signatures as needed to cover its pool of suspicious flows; it therefore is designed to minimize the number of signatures it generates. Our offline evaluation of Autograph on real DMZ traces reveals that the system can be tuned to generate *sensitive* and *specific* signature sets, that exhibit high true positives, and low false positives. Our simulations of the propagation of a Code-RedI-v2 worm demonstrate that by tattling to one another about port scanners they overhear, distributed Autograph monitors can detect worms earlier than isolated, individual Autograph monitors, and that the bandwidth required to achieve this sharing of state is minimal. DMZ-trace-driven simulations of the introduction of a novel worm show that a distributed deployment of 63 Autograph monitors, despite using a naive flow classifier to identify suspicious traffic, can detect a newly released Code-RedI-v2-like worm's signature before 2% of the vulnerable host population becomes infected. Our collected results illuminate the inherent tension between early generation of a worm's signature and generation of specific signatures.

Autograph is a young system. Several avenues bear further investigation. We are currently evaluating a single Autograph monitor's performance in an *online* setting, where the system generates signatures periodically using the most recent suspicious flow pool. Early results indicate that in a single signature generation interval, this online system can produce signatures for common HTTP worms, including Code-RedII and Nimda, and that using a minimal blacklist, the generated signatures can incur zero false positives. We will continue this evaluation using more diverse traces and protocol (port) workloads, to further validate these initial results. We look forward to deploying Autograph distributedly, including tattler, which has so far only been evaluated in simulation. Finally, we are keen to explore sharing information beyond port scanners' source IP addresses among monitors, in the interest of ever-faster and ever-higher-quality signature generation.

Acknowledgments

We are grateful to Vern Paxson of ICSI and to Casey Helfrich and James Gurganus of Intel Research for providing the DMZ traces used to evaluate Autograph. And to Adrian Perrig, Phil Gibbons, Robert Morris, Luigi Rizzo, and the anonymous reviewers, for insightful discussions and comments that improved our work.

Notes

[1]Signatures may employ more complicated payload patterns, such as regular expressions. We restrict our attention to fixed byte sequences.

[2]We include both poly- and metamorphism here; see Section 4.2.

[3]In future, worms may be designed to minimize the overlap in their successive infection payloads; we consider such worms in Section 4.2.

[4]Note that an IP address may have sent traffic before being identified as a scanner; such traffic will stored in the non-suspicious flow pool. We include only *subsequently* arriving traffic in the suspicious flow pool, in the interest of simplicity, at the expense of potentially missing worm traffic sent by the scanner before our having detected it as such.

[5]Worms that propagate very slowly may only accumulate in sufficient volume to be detected by Autograph for long values of t.

[6]Note that each Autograph monitor may independently choose its breakmark. Were the breakmark universal and well-known, worm authors might try to tailor payloads to force COPP to choose block boundaries that mix invariant payload bytes with changing payload bytes within a content block.

[7]We have since adopted a 16-byte COPP window in our implementation, to make it harder for worm authors to construct payloads so as to force particular content block boundaries; results are quite similar for $k = 16$.

[8]In cases where a source address owner complains that his address is advertised, the administrator of an Autograph monitor could configure Autograph not to report addresses from the uncooperative address block.

[9]We have implemented blacklists at this writing, but omit a full evaluation of them in the interest of brevity. Our experience has shown that blacklists of even 2 to 6 disallowed signatures can significantly reduce false positives caused by misclassified innocuous flows, for HTTP traffic.

References

[1] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A. Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC) 20*, 8 (Oct. 2002).

[2] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003).

[3] CISCO SYSTEMS. Network-Based Application Recognition. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newf%t/122t/122t8/dtnbarad.htm`.

[4] DSHIELD.ORG. DShield - Distributed Intrusion Detection System. `http://dshield.org`.

[5] JUNG, J., PAXSON, V., BERGER, A. W., AND BALAKRISHNAN, H. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2004).

[6] KREIBICH, C., AND CROWCROFT, J. Honeycomb—Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)* (Nov. 2003).

[7] LEMOS, R. Counting the Cost of Slammer. CNET news.com. `http://news.com.com/2100-1001-982955.html`, Jan. 2003.

[8] MOORE, D., AND SHANNON, C. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the 2002 ACM SIGCOMM Internet Measurement Workshop (IMW 2002)* (Nov. 2002).

[9] MOORE, D., SHANNON, C., VOELKER, G. M., AND SAVAGE, S. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of IEEE INFOCOM 2003* (Mar. 2003).

[10] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Oct. 2001).

[11] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (Dec. 1999).

[12] PROVOS, N. A Virtual Honeypot Framework. Tech. Rep. 03-1, CITI (University of Michigan), Oct. 2003.

[13] RABIN, M. O. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[14] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RFC 1889 - RTP: A Transport Protocol for Real-Time Applications, Jan. 1996.

[15] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. The Early-Bird System for Real-time Detection of Unknown Worms. Tech. Rep. CS2003-0761, UCSD, Aug. 2003.

[16] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm Fingerprinting. Unpublished draft, received May 2004.

[17] STANIFORD, S., HOAGLAND, J. A., AND MCALERNEY, J. M. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security 10*, 1-2 (Jan. 2002).

[18] STANIFORD, S., PAXSON, V., AND WEAVER, N. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).

[19] THE SNORT PROJECT. Snort, The Open-Source Network Intrusion Detection System. `http://www.snort.org/`.

[20] UNIVERSITY OF OREGON. University of Oregon Route Views Project. `http://www.routeviews.org/`.

[21] WEAVER, N. C. Warhol Worms: The Potential for Very Fast Internet Plagues. `http://www.cs.berkeley.edu/~nweaver/warhol.html`.

[22] WU, J., VANGALA, S., GAO, L., AND KWIAT, K. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the Network and Distributed System Security Symposium 2004 (NDSS 2004)* (Feb. 2004).

[23] YEGNESWARAN, V., BARFORD, P., AND JHA, S. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of Network and Distributed System Security Symposium (NDSS 2004)* (Feb. 2004).