Exploiting Temporal Consistency to Reduce False Positives in Host-Based, Collaborative Detection of Worms

David J. Malan and Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts, USA
{malan,smith}@eecs.harvard.edu

ABSTRACT

The speed of today's worms demands automated detection, but the risk of false positives poses a difficult problem. In prior work, we proposed a host-based intrusion-detection system for worms that leveraged collaboration among peers to lower its risk of false positives, and we simulated this approach for a system with two peers. In this paper, we build upon that work and evaluate our ideas "in the wild." We implement Wormboy 2.0, a prototype of our vision that allows us to quantify and compare worms' and non-worms' *temporal consistency*, similarity over time in worms' and non-worms' invocations of system calls. We deploy our prototype to a network of 30 hosts running Windows XP with Service Pack 2 to monitor and analyze 10,776 processes, inclusive of 511 unique non-worms (873 if we consider unique versions to be unique non-worms). We identify properties with which we can distinguish non-worms from worms 99% of the time. We find that our collaborative architecture, using patterns of system calls and simple heuristics, can detect worms running on multiple peers. And we find that collaboration among peers significantly reduces our probability of false positives because of the unlikely appearance on many peers simultaneously of non-worm processes with worm-like properties.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Invasive software*

General Terms

Algorithms, Experimentation, Measurement, Security

Keywords

collaborative detection, HIDS, IDS, host-based intrusion detection, native API, peers, system calls, system services, temporal consistency, Win32, Windows, worms

1. INTRODUCTION

No longer in hours but in minutes are worms' rates of propagation now measured [23,31,45]. So fast are the fastest that human intervention no longer is possible [41,42].

Automated detection is necessary. But with automation comes a risk of false positives, whereby benign applications (non-worms) are misclassified as worms. Inherent in intrusion-detection systems (IDSes), after all, are "virtual knobs." Settings that detect many behaviors (and thus many worms) also tend to produce many false positives. Settings more tailored to known worms' behaviors produce fewer false positives but are easier for worms' authors to circumvent in future designs. Ideal is an IDS that detects many behaviors without producing false positives. We pursue that ideal in this work. With collaboration among hosts, we are able to reduce the risk of false positives in host-based detection of worms.

Worms can be distinguished from non-worms by their simplicity and periodicity: their design is to spread, and their execution is cyclical. Of course, even non-worms can manifest cyclical behavior reminiscent of worms', but we are less likely to see such behavior simultaneously on networked, but otherwise independent, hosts, unless it's on purpose. Worms' actions are so relatively few that we are more likely to detect them in near lockstep on multiple peers than the actions of more complicated applications with many more code paths. We can therefore lower the risk of false positives in worms' detection by monitoring the collective behavior of many hosts for similarities. Our goal is to avoid misclassifying non-worms that might otherwise look like worms from the perspective of a single host.

To validate these claims, we proposed, in prior work [21], an IDS for worms that leverages collaboration among hosts to lower its risk of false positives. Specifically, we envision a network of peers (Figure 1), each running software that takes *snapshots* (*i.e.*, produces summaries) of its processes' behavior during some window of time. On some schedule, peers submit those snapshots for analysis to some *snapshot server*, a supernode responsible for a network of peers. Treating snapshots like nodes in a graph, the supernode searches those submissions for cliques of similarity, whereby pairs of snapshots, if found to be "similar" according to some measure, are treated as edges.[1] If the cooperative's supernodes determine that peers' behavior is *anomalous* (because the size of a clique exceeds some threshold), a worm is assumed present, and a response can be initiated.

Through simulations with traces of 9 variants of worms and 25 non-worms, we found in prior work [21] that two peers, upon exchanging summaries of system calls recently executed, can decide that they are, more likely than not, both executing the same worm between 76% and 97% of the time.

In this work, we build upon those results and evaluate our ideas "in the wild." We implement and deploy a prototype of our vision, Wormboy 2.0, to a network of 30 hosts running Windows XP with Service Pack 2. We use our implementation to monitor and analyze 10,776 processes, inclusive of 511 unique non-worms (873 if we consider unique versions to be unique non-worms). We investigate a host-based, collaborative architecture's probabilities of true positives and false positives, focusing on the latter.

As a result of this work, we identify properties that distinguish worms from non-worms. Using those properties, our architecture accurately classifies 99% of processes as non-worms. We also find that our collaborative architecture, using patterns of system

[1]We propose cliques, despite their hardness, for the sake of discussion, as they perfectly capture k-wise similarity among k peers. In practice, approximations should suffice for our purposes (Section 5.2).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WORM'06, November 3, 2006, Fairfax, Virginia, USA.
Copyright 2006 ACM 1-59593-551-7/06/0011 ...$5.00.

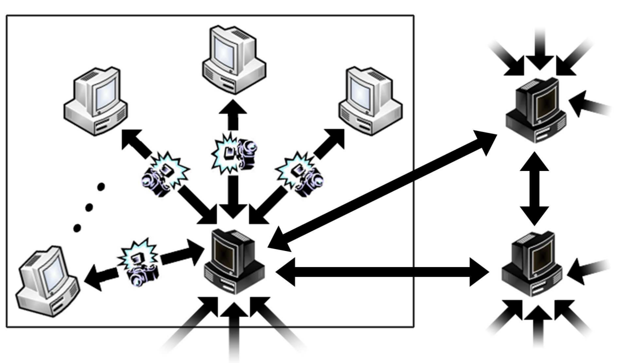


Figure 1: Our vision of collaborative detection of fast-spreading worms using collaborative networks. Colored black are three *snapshot servers*, supernodes responsible for networks of (gray-colored) peers. On some schedule, peers submit *snapshots* (*i.e.*, summaries) of their internal behavior to a snapshot server that then exchanges information with other supernodes. Too many similarities within or across networks suggest *anomalous behavior* (*i.e.*, a worm's presence). Boxed is one such network, implemented with `WORMBOY.{EXE,SYS}` and `WormboyD`.

calls and simple heuristics, can detect worms running on multiple peers. And, because of the unlikely appearance on many peers simultaneously of non-worm processes with worm-like properties, we find that collaboration among peers significantly reduces our probability of false positives.

In the section that follows, we elaborate on our vision and earlier results. In Section 3, we formulate this work's research questions. In Section 4, we describe Wormboy's role in this work and offer implementation details. In Section 5, we validate our vision's efficacy: we quantify the number of non-worm processes that might, if not handled carefully, be mistaken for worms by a collaborative architecture; we establish empirically that a collaborative network can detect processes with similar behavior running on multiple hosts; and we demonstrate that collaboration among peers reduces our probability of false positives. In Section 6, we discuss threats to host-based, collaborative detection of worms. Finally, in Sections 7 and 8, respectively, we explore related work and conclude.

2. COLLABORATIVE DETECTION

In prior work [21], we investigated techniques for modeling and quantizing worms' and non-worms' behavior. Using those techniques, we proposed a new definition of anomalous behavior in a network for collaborative detection. In this section, we summarize our prior work. In Section 2.1, we review our model for behavior. In Section 2.2, we restate our definition of anomalous behavior. In Section 2.3, we recognize that a worm's execution on one peer will not likely be perfectly synchronized with that worm's execution on another peer; we present our technique for recognizing that both executions exhibit similar behavior and belong to the same executable.

2.1 Behavior as Snapshots

We model processes' behavior as patterns of system calls, and we quantize it with *snapshots*, sets of system calls executed during some window of time. Formally, a snapshot is an unordered set of the form $S = (s_0, s_1, \ldots, s_{n-1})$, where each s_i represents a system call that was invoked one or more times during some window of time. In prior work [21], we found unordered sets more tolerant of variation in worms' execution than sets ordered according to calls' relative frequencies of invocation. This definition also facilitates detection of worms whose authors employ randomization along code paths, a threat we revisit in Section 6.

Insofar as system calls circumscribe kernel space, restricting execution's passage from Ring-3 to Ring-0, they facilitate summarization of code into low-level, but semantically cogent, building blocks. Though other models are possible, system calls offer a useful proxy for behavior [8,16,29,39,40]. We select them for the

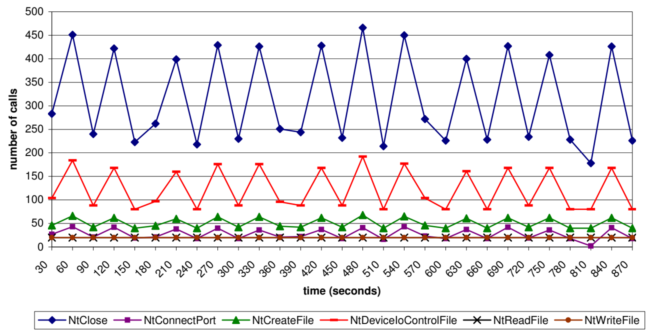


Figure 2: Calls into kernel space by I-Worm/Sasser.B, representative of worms' tendency toward simplicity and periodicity. Point (i, j) indicates j invocations of some system call between times i and $i+30$. Omitted for visual clarity are similar patterns of invocations of other system calls. Windows of 30 seconds are sufficient to capture this worm's cycles [21].

sake of current exploration; our vision's modular design allows for substitution of them with alternative proxies in the future.

In that snapshots comprise an aggregate (*i.e.*, window's worth) of system calls, they allow us to capture patterns in worms' execution. Figure 2, for instance, illustrates I-Worm/Sasser.B's consistency over time of calls into kernel space. The figure plots the numbers of invocations of system calls during 30-second windows of time. We experimented in prior work [21] with windows of 5, 15, 30, and 60 seconds, ultimately finding 5 seconds too brief to capture worms' periodicity and 60 seconds unnecessarily long. Neither 15 nor 30 proved consistently better than the other; we utilize both in this paper but, for now, do not further investigate either's superiority.

2.2 Redefining Anomalous Behavior

We define *anomalous behavior* as we do snapshots: in terms of patterns of system calls. But we promote an atypical definition of *anomalous*. Host-based IDSes tend to evaluate a host's actions vis-à-vis prior actions or blacklisted actions: a host's behavior is deemed anomalous if it differs from that host's prior actions or a pre-determined list of blacklisted actions. We, in contrast, eschew reliance on history and aspire instead to generalize the problem of worms' discovery away from recognizance of pre-determined actions (and pre-defined signatures) toward more generalized detection of widespread and coordinated behavior. We offer an alternative definition of anomalous behavior: a host's behavior is anomalous if it correlates all too well with other networked, but otherwise independent, hosts' behavior. We argue that anomalous behavior, triggered by some worm, can be detected because of worms' *temporal consistency*, which we define as low temporal variance (*i.e.*, similarity) in invocations of system calls.

2.3 Similarity as Temporal Consistency

Execution of some worm might not be perfectly synchronized within some network of peers, particularly if hosts become infected at different times. Any measure of snapshots' similarity must therefore tolerate offsets in timing. In prior work [21], we evaluated two such measures. We adopt for our current work the superior of the two, which proved tolerant not only of offsets in timing but also of differences in hosts' speeds and configurations.

We judge the similarity of two snapshots, S_1 and S_2, by way of $S_1 \cap S_2$. Specifically, we define the percentage of similarity between two snapshots as $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)}$, which is effectively a measure of the number of system calls common to both snapshots. If $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)} \geq 0.5$ for τ percent of pairs of snapshots over time, the process to which the snapshots pertain is said to be *temporally consistent* with *degree* τ. Informally, $\tau \geq 0.5$ implies that two processes behave "mostly the same most of the time." This rate, τ, is thus the probability with which two peers, upon

	15 s	30 s
I-Worm/Bagle.Q	76%	81%
I-Worm/Bagle.S	76%	73%
I-Worm/Mydoom.D	97%	93%
I-Worm/Mydoom.F	97%	93%
I-Worm/Jobaka.A	97%	93%
I-Worm/Sasser.B	97%	93%
I-Worm/Sasser.D	97%	93%
Worm/Lovesan.A	97%	93%
Worm/Lovesan.H	97%	93%

Table 1: Probability, τ, with which two peers, upon exchanging snapshots of their internal behavior, can decide using intersection of snapshots that they are, more likely than not, both executing the same worm during some window of time, for windows of 15 and 30 seconds, per our prior work [21]. In other words, percentages of all possible pairs of samples from some worm for which $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)} \geq 0.5$, where S_1 and S_2 are snapshots, unordered sets of system calls executed during some window of time. Consider, for example, Worm/Lovesan.H: 97% of its snapshots look "mostly the same," using 15-second windows.

exchanging snapshots of their internal behavior, can decide using intersection of snapshots that they are, more likely than not, both executing the same worm during some window of time.

Blind as this measure of similarity is to system calls' order of invocation, it allows for the emergence of patterns despite slight differences in execution. It was with this measure that we detected with near certainty ($\tau = 97\%$) in prior work [21], using windows of 15 or 30 seconds, every one of the worms in our earlier study save one (Table 1), but even the latter's τ was 76%.[2] Moreover, of the 25 non-worms in that study, only two boasted traces with $\tau > 90\%$; for both did we propose simple filtration to avoid the false positives.

3. RESEARCH QUESTIONS

To validate the efficacy of host-based, collaborative detection "in the wild," we focus in this work on these questions.

1. *Are non-worms, like worms, temporally consistent?* If so, we must identify properties that distinguish the two.
2. *Can we detect processes with similar behavior on multiple hosts?* If so, we can detect the outbreak of a worm, the behavior of which across hosts is likely to be similar: bounded by time as are fast-spreading worms by definition, there are only so many ways for them to achieve some effect on a host *quickly*. A worm's filename and executable, by contrast, can be too easily altered during propagation (as through metamorphosis) and are, thus, less reliable than more dynamic techniques. Presumably, the more "worm-like" a process (*i.e.*, the more temporally consistent), the more likely we are to detect it if running on multiple hosts.
3. *Can we avoid mistaking popular non-worms for worms?* We cannot assume that processes common to many hosts (*e.g.*, `explorer.exe`) are necessarily worms, lest we infer incorrectly that an attack is in progress. And we should not confuse a non-worm running on one host with a worm running on another, even if behaving similarly, lest we overstate an outbreak's severity.

To answer these questions, we transition from simulation to actual implementation of a prototype of our vision: Wormboy 2.0. We deploy this prototype to dozens of hosts running hundreds of non-worms in order to evaluate collaborative detection's efficacy "in the wild" through collection and analysis of real-world data. In the section that follows, we offer implementation details for Wormboy 2.0 to make clear our data's origins and manner of collection.

[2]We called the worms by their names according to AVG Free Edition 7.0.322 [9].

4. IMPLEMENTATION

Having defined hosts' behavior in terms of snapshots, we implemented Wormboy, a prototype of our host-based, collaborative system with one purpose: to collect and analyze snapshots. Ultimately the foundation for a worm-focused IDS, Wormboy includes both client and server sides that, together, implement a network of peers per our vision (Figure 1). With one such network alone, we are able to collect and analyze data from real-world hosts.

Wormboy's target is Windows, as the platform offers a richness of available worms and is a perpetual recipient of new attacks. As its proxy for processes' behavior, Wormboy uses Windows's *system services* (its "native API"), the nearest equivalents of Linux's and UNIX's system calls.[3]

A link to Wormboy's source code appears at this paper's end.

4.1 Wormboy on the Client Side

On the client side, Wormboy is implemented as a cooperation between a kernel-mode driver (`WORMBOY.SYS`) and a user-mode application (`WORMBOY.EXE`), both written in C. Inspired by work by Sabin [34], Nebbett [24], Harris [12], and Dabak *et al.* [5], Wormboy's client currently supports Windows XP with Service Pack 2.[4] Upon load, `WORMBOY.SYS` hooks all but three system services.[5,6] These hooks effectively log (to unpaged memory) each invocation of a system service during some window of time, capturing not only the call's service ID but also the caller's PID and path. At the end of each window, `WORMBOY.EXE` polls `WORMBOY.SYS` for that window's snapshots, the structure of which appears in Figure 3. The application then transmits those snapshots to Wormboy's snapshot server via XML-RPC [13, 47].

4.2 Wormboy on the Server Side

On the server side, Wormboy's snapshot server is implemented in Java as a XML-RPC listener (`WormboyD`). Upon receipt of a window's worth of snapshots from Wormboy clients, `WormboyD` analyzes the structures for similarities exceeding specified thresholds of interest (Section 5). At present, the server simply logs the results of its analysis to disk for human review. In our work's next phase will we add support for the exchange of snapshots with other supernodes in order to investigate questions of scalability and efficiency based on the current work's results.

[3]Depending on the version of Windows, Windows's native API comprises between 211 and 295 system services [28], implemented in kernel space by `NTOSKRL.EXE` and exposed with stubs in user space by `NTDLL.DLL`, against which most higher-level Win32 APIs are dynamically linked. When called to invoke a system service, a stub in `NTDLL.DLL` invokes `SharedUserData!SystemCallStub` after moving into register `EAX` the service's *service ID* and into register `EDX` a pointer to the call's arguments. To trap from user-mode to kernel-mode, `SharedUserData!SystemCallStub` then executes Intel's `SYSENTER` instruction (for the Pentium II and newer) or AMD's `SYSCALL` instruction (for the K7 or newer); on older CPUs, `SharedUserData!SystemCallStub` executes a slower `INT 2e` instruction. Control is ultimately passed to `_KiSystemService`, which dispatches control to the appropriate service by indexing into `_KeServiceDescriptorTable` for the service's address and number of parameters using the value in `EAX`. [5, 10, 11, 14, 27, 32]

[4]With minor modifications could Wormboy's client also support Windows NT (with and without Service Packs 3 through 6), Windows 2000 (with and without Service Packs 1 through 4), Windows XP (with and without Service Packs 1 and 2), and Windows 2003 Server (with and without Service Pack 1).

[5]Wormboy inserts trampolines into `_KeServiceDescriptorTable`; by default, `_KeServiceDescriptorTable` is read-only, so the driver first disables the `WP` bit in register `CR0` [30, 38].

[6]Windows seems to make certain assumptions about system services `NtContinue` and `NtRaiseException`, whereby it attempts to manipulate the stack frame based on register `EBP` [33]; inasmuch as our hooks insert a frame of their own, we do not hook these services to avoid blue screens. Nor do we hook `NtQueryInformationProcess`, as our other hooks invoke undocumented features of this service themselves.

Benchmark	Calls	Runtime		Overhead
		w/o	w/	
Adobe Photoshop 7.0.1	258,549	1574 s	1589 s	0.95%
Adobe Premiere 6.5	13,379,755	1830 s	1864 s	1.9%
Ahead Software Nero Express 6.0.0.3	46,869,089	2536 s	2610 s	2.9%
Microsoft Office XP SP2	2,317,059	1054 s	1065 s	1.0%
Microsoft Windows Media Encoder 9.0	1,672,449	2141 s	2164 s	1.1%
Mozilla 1.4	51,956,045	2883 s	3002 s	4.1%
MusicMatch Jukebox 7.10	308,793	2680 s	2699 s	0.71%
Roxio VideoWave Movie Creator 1.5	2,287,867	1553 s	1569 s	1.0%
WinZip Computing WinZip 8.1	4,775,630	1704 s	1717 s	0.76%

Table 2: Results of executing PC World's WorldBench 5 [26] benchmarking suite without (w/o) and with (w/) Wormboy running on a 550MHz Pentium III with 384MB RAM atop Windows XP with Service Pack 2, averaged over ten runs of the suite, the standard deviations for which varied from 4 to 23 seconds. Wormboy's average impact on runtime did not exceed 4.1%.

```
typedef struct {
    ULONG counts[NUM_SERVICES];
    ULONG pid;
    CHAR  path[MAX_PATH];
} snapshot;
```

Figure 3: Wormboy's definition of a *snapshot*. On the client-side, Wormboy's kernel-mode driver maintains in non-paged memory one of these structures for each live process. On some schedule, Wormboy's user-mode application polls the driver for those structures (after which the driver zeroes `counts`) and marshals them over XML-RPC to Wormboy's snapshot server for analysis.

4.3 Performance

Though further optimization of our implementation is possible, current performance is promising. Wormboy's client-side impact on peers' runtime does not exceed 4.1% (Table 2).

With calls into kernel space as our proxy for behavior, performance of Wormboy's client is of particular import, lest our hooking of as many as thousands of calls per second interfere with hosts' actual work. Not only, then, does Wormboy log calls to unpaged memory, it also executes few instructions to perform its logging.

In future work will we consider the performance of Wormboy on the server side. In particular, we will investigate the efficiency with which supernodes can analyze snapshots for similarities.

5. RESULTS

We present in this section the results of Section 3's inquiries. To answer those questions using real-world data, we deployed Wormboy to a network of 30 heavily-used, independent hosts (spread across domains throughout North America) running Windows XP with Service Pack 2 for 24 hours. With this deployment were we ultimately able to monitor and analyze 10,776 processes, inclusive of 511 unique non-worms (873 if we consider unique versions to be unique non-worms).

Though these hosts, as real systems on the Internet, were by nature exposed to worms, we did not inject worms into this network ourselves. Nor did we set out to detect actual worms for this paper; we focus herein on non-worms and the avoidance of false positives. In earlier work [21], we examined the temporal consistency of actual worms and our probability of true positives.

We present this work's results in turn. We first quantify the number of non-worm processes that a collaborative network might tend to mistake for worms, were it not for certain properties unique to the latter (Section 5.1). We then demonstrate empirically, using non-worms with worm-like properties as proxies for worms, that a collaborative network can, in fact, detect worms executing on multiple hosts (Section 5.2). Finally, we show how collaboration among peers reduces our probability of false positives (Section 5.3).

5.1 Identifying τ, r, and r'

In prior work [21], we investigated the degree to which 25 non-worms in two benchmarking suites were temporally consistent. Through simulation, we found that only two of those 25 (8%) boasted traces for which $\tau > 90\%$ using windows of 15 seconds;

15 s	30 s
	1.EXE
aexplore.exe	aexplore.exe
aolsoftware.exe	aolsoftware.exe
ApntEx.exe	ApntEx.exe
BESClient.exe	BESClient.exe
	ccApp.exe
CCPROXY.EXE	CCPROXY.EXE
cvpnd.exe	cvpnd.exe
explorer.exe	explorer.exe
iexplore.exe	iexplore.exe
ntbackup.exe	ntbackup.exe
OUTLOOK.EXE	OUTLOOK.EXE
QCWIZARD.EXE	QCWIZARD.EXE
SNDSrvc.exe	
sshd.exe	sshd.exe
ViewMgr.exe	ViewMgr.exe
war3.exe	war3.exe
	wmplayer.exe
	WRSSSDK.exe

Table 3: Nineteen non-worms that exhibit worm-like behavior, for windows of 15 and 30 seconds. Of the 511 unique non-worms in our study, we might mistakenly classify as worms just 15 (2.9%) using windows of 15 seconds and 18 (3.5%) using windows of 30 seconds. Processes common to both windows are aligned for visual clarity.

only one of the 25 boasted a trace for which $\tau > 90\%$ using windows of 30 seconds. With simple heuristics did we then distinguish those non-worms from worms, despite their apparent similarity. For instance, we considered for each process not only its τ but also its rate of calls, r, into kernel space. In particular, one potential false positive averaged no more than $r = 1$ call into kernel space per second, whereas even our "slowest" of worms, I-Worm/Jobaka.A, averaged $r = 64$ calls per second. Insofar as processes averaging nearly zero calls per second do not likely belong to fast-spreading worms, we required that large τ, to be worrisome, be accompanied by non-negligible rates of calls (*e.g.*, $r \geq 64$).

By way of analysis with Wormboy 2.0 of not 25 but thousands of processes, we have since found it advantageous to identify an additional property besides τ and r that distinguishes worms from non-worms. For each of the processes for which `WormboyD` received snapshots over the course of 24 hours, we reviewed up to an hour's worth of data, exhaustively measuring the similarity of each snapshot received during that frame against every other snapshot received during the same. No matter our windows' size, we find that at least 315 of the 10,776 non-worm processes boast $\tau \geq 76\%$ and $r \geq 64$. Those 315 processes belong to 85 (17%) of our 511 unique non-worms.

But if we further require that some process actually utilize the network at a rate, r', no slower than that of our slowest of worms, we fare even better. (For reasons of privacy, Wormboy 2.0 does not capture hooked calls' parameters, the implication of which is that we can only estimate r' for now during automated analysis.[7]) We now find, using a window of 15 seconds, that only $145/10,776 \approx 1\%$ of processes appear to be worms and, equiv-

[7]Because Wormboy 2.0 does not capture hooked calls' parameters, we cannot detect network activity with certainty using ser-

	15 s	**30 s**
Non-Worm Processes	10,776 (511)	10,776 (511)
. . . w/ $\tau \geq 76\%$, $r \geq 64$	351 (77)	315 (85)
. . . w/ $\tau \geq 76\%$, $r \geq 64$, $r' > \delta$	145 (15)	112 (18)

Table 4: Results of exhaustive, worst-case examination of 10,776 non-worm processes for worm-like behavior, where τ denotes a process's degree of temporal consistency, r denotes a process's rate of calls to system services, r' denotes a process's rate of network activity, and δ denotes a threshold (the slowest rate of network activity witnessed among our 9 worms). Listed parenthetically are the numbers of *unique* non-worms (irrespective of version) to which processes belong. With intelligent filtration, as few as 15 (2.9%) of 511 unique non-worms resemble worms.

alently, that 99% of processes appear not to be worms. And those 145 processes belong to just 15 (2.9%) of our 511 unique non-worms (Table 3). We summarize these results in Table 4. Though we earlier found through simulation 8% (2 of 25) non-worms to resemble worms, we now lower that bound to 1%, using real-world data filtered not only by τ and r but also by r'. Other filters are certainly possible. But that only 15 of 511 remain after these filters alone reinforces the potential of collaborative detection, insofar as so few out of hundreds of non-worms might potentially evince worm-like behavior on many hosts at once.

5.2 Detecting Processes across Peers

Prior work [21] suggests that worms can be detected on multiple hosts because of worms' degrees of temporal consistency (*e.g.*, $\tau > 90\%$), and current work suggests that certain "worm-like" non-worms, if not properly filtered, might be detected as well. We confirm that hypothesis in this section. In particular, we look for positive correlation between some process's τ and the probability with which our collaborative network recognizes that process's execution on multiple peers. Rather than inject worms into our network of 30 hosts, we look to our most worm-like of non-worms (Section 5.1) as proxies for worms. For the purposes of this inquiry, we treat those non-worms with particularly high τ as representative of worms. We expect that large τ should imply high rates of recognition, whereas the smallest of τ should imply few, if any, matches in snapshots from peers.

If we examine each of our 30 peers' non-worms over 24 hours, we find that only for large τ are multiple peers likely to recognize a common process. Figure 4 depicts this result, plotting non-worms' rates of recognition against non-worms' degrees of temporal consistency. We define *rate of recognition* as follows: if some non-worm is executing during some window on $n \geq 2$ peers, and we determine that m such instances are similar, then that non-worm's rate of recognition for that window is said to be m/n. By similar, we mean that, for each pair of processes among the m, $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)} \geq 0.5$, where S_1 and S_2 are snapshots, for at least 76% (our prior work's least worrisome τ) of the snapshots submitted for the processes (over the course of an hour). Informally, we deem two non-worms similar if at least 76% of their snapshots look "mostly the same." In terms of cliques, a rate of recognition of m/n for some process during some window implies recognition of an m-clique of similarity among snapshots from *all* of our peers, n of which are actually executing that process. (It is not necessarily the case that an n-clique also exists during that window, as processes with $\tau < 100\%$ might not "look the same" across all peers during some window.) Because no cliques in our study exceeded $m = 6$, we compiled our results for Figs. 4 and 5 using brute-force analysis. Cooperative networks boasting larger n (and, in turn, larger m) will demand more efficient approaches;

vice IDs alone. We thus infer possible network activity from frequent calls to `NtOpenFile` (the service involved in sockets' creation), `NtDeviceIoControlFile` (the service involved in packets' transmission), and `NtCloseFile` (the service involved in sockets' termination) [17]. With the value of these three services in filtration now clear, we will examine arguments at least to those services in future implementations to detect with certainty network (as opposed to, say, file) activity.

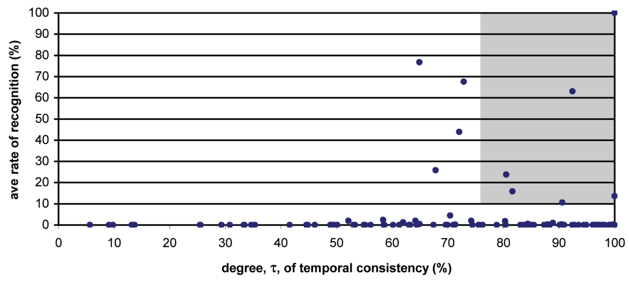


Figure 4: Rates of recognition of non-worms as a function of those non-worms' degrees, τ, of temporal consistency, averaged over up an hour's worth of activity during 24 hours of analysis using windows of 30 seconds. If some non-worm is executing during some window on n peers, and we determine that m such instances are similar, then that non-worm's rate of recognition is said to be m/n. All non-worms depicted boasted rates of calls into kernel space, r, greater than 64 per second (the rate of our slowest of worms). As we would hope for actual worms, only processes with large τ are detected with non-negligible probability. Dots representing large τ but low rates of detection tend to belong to short-lived processes that, because of their brevity, tend not to appear among our 30 hosts simultaneously, unlike worms. Shaded is this figure's upper-right quadrant, which includes six non-worms with $\tau \geq 76\%$ that were detected at least 10% of the time. We expect actual worms to fall within this quadrant as well, per prior work [21].

we expect, as in other domains, that approximations (as with Bloom filters or randomization) will suffice in future work.

To be clear, n is not necessarily our network's size but, rather, the number of hosts on the network executing some non-worm. As such, m/n is simply a rate of recognition, not a rate of infection.

Though peers' average rates of recognition are not strictly correlated with rising τ, in no 30-second window during our 24 hours of data do multiple peers detect processes common to them if those processes' τ are below 65%. For $\tau \geq 65\%$, we detect common processes at non-negligible rates, except for short-lived processes (whose points fall on Figure 4's x-axis) that, because of their brevity, tend not to appear among our 30 hosts simultaneously. Our goal, though, is to detect fast-spreading worms, whose activity, by nature, is more likely to be ongoing than brief. That processes with $\tau \geq 65\%$ are, in fact, recognized across peers reinforces host-based, collaborative detection's potential, inasmuch as τ for every one of our worms in prior work [21] was at least 76%. Because of worms' relatively high degrees of temporal consistency, we expect they will fall within Figure 4's shaded, upper-right quadrant, as do six of our most worm-like non-worms.

5.3 Avoiding False Positives

Because our goal is to detect worms rapidly (*e.g.*, within a single, 30-second window), it is necessary to examine not only non-worms' average rates of recognition but also their worst-case, maximal rate of recognition (*i.e.*, the maximum of m/n seen over time). After all, even if some non-worm goes undetected most of the time, a single window's worth of similar behavior across many peers might induce a false positive, whereby we judge that non-worm a worm. Figure 5 contrasts average and maximal rates of recognition for those non-worms whose average rates of recognition exceed 1%.

Particularly worrisome are those non-worms whose maximal rates of recognition are $50\% < m/n \leq 100\%$, the result of which is that, on occasion, those non-worms were detected on most, if not all, of the hosts on which they were running. But in none of those cases were the non-worms running on most of the peers in our network. In fact, in none of these cases was m (or n) greater than 4, whereas our network consisted of 30 peers, an apparent "infection" rate, ι, of $4/30 \approx 13\%$. Accordingly, provided we set our threshold for detection at 13% (*i.e.*, require, for a worm to be assumed present, that some process appear similar on $\iota > 13\%$ of peers), our cooperative of 30 peers avoids a false positive. In other words, a high rate of recognition (m/n) does not imply

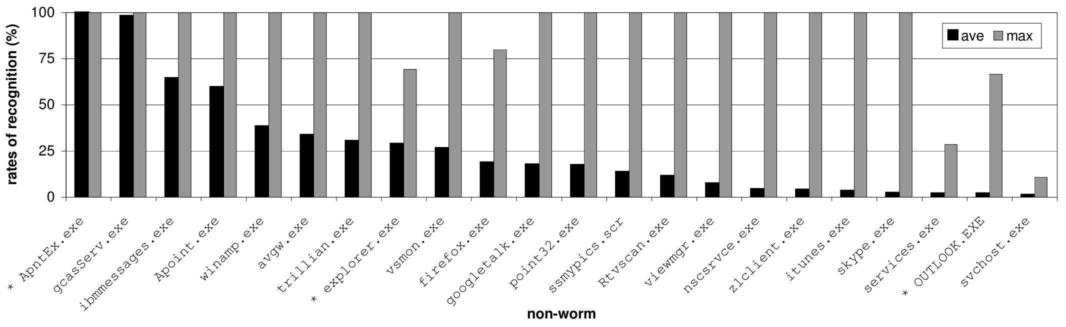


Figure 5: Average and maximal rates of recognition for non-worms whose average rates of recognition exceed 1%. Figure 4 plots these same average rates against non-worms' degrees of temporal consistency. Only three of these non-worms (*) are worrisome in that they also appear in Table 3, boasting worm-like τ, r, and r'.

a high rate of infection or, rather, in the case of non-worms, a likelihood of false positives.

Based on these results, we propose, for now, $\iota = 13\%$ as a threshold for host-based, collaborative detection: if a worm-like process (*i.e.*, with worrisome r and r') appears on more than 13% of peers in a network, a worm shall be assumed present. We will vet this parameter's reliability in future work.

Our collaborative network's potential for false positives is indeed less than Figure 5 suggests. If we cross-reference those non-worms in Figure 5 with those in Table 3, we find that only three are "worm-like," insofar as they appear in both: `ApntEx.exe`, `explorer.exe`, and `OUTLOOK.EXE`. Filtration by τ, r, and r' therefore limits our risk of false positives to the actions of just three of 511 non-worms. At least two of these non-worms (`iexplore.exe` and `OUTLOOK.EXE`) do involve frequent network activity, but not so frequent as our fastest of worms [21]. Moreover, it may, in fact, prove feasible to whitelist these most popular of non-worms (as with read-only hashes of their executables). Our focus for now is on our more generalized techniques.

Of course, not only might false positives induce an IDS to infer incorrectly that an attack is in progress, they might also induce an IDS to overstate an actual outbreak's severity. By confusing non-worms with actual worms, an IDS might conclude that more hosts are infected than actually are, the result of which might be a premature or unnecessarily severe reaction, depending on the IDS's mechanism for response.

To determine the likelihood with which non-worms resemble might actual worms, we performed an exhaustive comparison of snapshots from our 19 most worm-like non-worms (15 of which appeared worm-like using windows of 15 seconds and 18 of which appeared worm-like using windows of 30 seconds) among our 511 unique non-worms (Table 3) with snapshots from our first study's 9 worms. The results are striking: 14 of the 19 non-worms are similar to actual worms. More formally, the percentage of all possible pairs of snapshots for which $\frac{|S_1 \cap S_2|}{\max(|S_1|, |S_2|)} \geq 0.5$ itself exceeds 50% for these 14 non-worms and is even as high as 100% for one (Table 5).

Closer examination of these non-worms' and worms' snapshots offers some insight. If we consider, for instance, the most striking of these matches, `sshd.exe` vis-à-vis I-Worm/Mydoom.F, we see that, while the two manifest remarkable overlap in services utilized, their frequency distributions are markedly different (Figure 6), the implication of which is that consideration of either in filtration might, in fact, prove useful in such cases.

But far simpler it is to filter based on non-worms' rates, r', of network activity. Because Wormboy 2.0 does not capture hooked calls' parameters (again, for reasons of privacy), our current implementation can only estimate those rates. In future deployments will we accurately assess processes' r'. For the work at hand, we resort to manual analysis of these worrisome matches and find that none boasts as high rates of sockets' creation, utilization, and termination as do their matched worms.

Non-Worm	Worm(s)	Similarity
`1.EXE`	I-Worm/Mydoom.F	58%
`aolsoftware.exe`	I-Worm/Sasser.D	74%
`apntex.exe`	Worm/Lovesan.A	56%
`besclient.exe`	I-Worm/Mydoom.F	97%
`ccproxy.exe`	I-Worm/Mydoom.F	77%
`cvpnd.exe`	I-Worm/Sasser.D	73%
	I-Worm/Jobaka.A	66%
	I-Worm/Sasser.B	64%
`explorer.exe`	I-Worm/Mydoom.F	85%
	I-Worm/Bagle.S	61%
	I-Worm/Bagle.Q	60%
	Worm/Lovesan.H	58%
`iexplore.exe`	I-Worm/Mydoom.F	85%
	Worm/Lovesan.A	71%
	I-Worm/Mydoom.D	54%
`OUTLOOK.EXE`	I-Worm/Mydoom.F	64%
`SNDSrvc.exe`	Worm/Lovesan.H	97%
	I-Worm/Sasser.D	72%
`sshd.exe`	I-Worm/Mydoom.F	100%
`ViewMgr.exe`	I-Worm/Sasser.D	74%
	I-Worm/Mydoom.F	67%
`wmplayer.exe`	I-Worm/Mydoom.F	51%
`WRSSSDK.exe`	I-Worm/Mydoom.F	89%

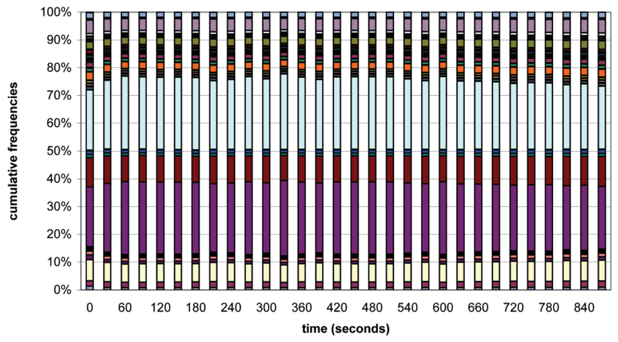
Table 5: Similarity over time of 14 worm-like non-worms (Table 3) with actual worms, determined using windows of 30 seconds. The striking similarities suggest that further reduction of a collaborative system's probability of false positives requires further refinements in filtration (*e.g.*, some consideration of calls' order or relative frequencies).

6. THREATS

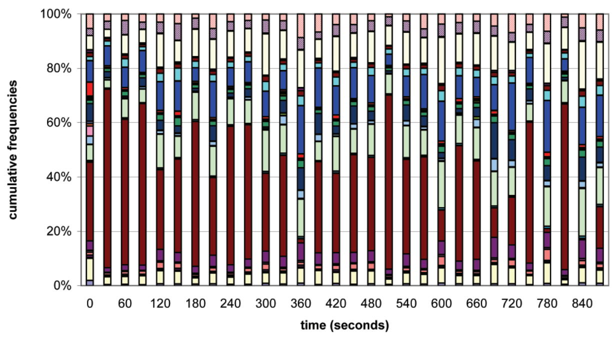
As with most host-based defenses, adversaries tend to adapt to the latest heuristics. Our vision, like others, certainly comes with its own risks. Worms designed to vary the frequencies of their calls into kernel space are perhaps the most obvious threat to our vision's design. Superfluous calls to system services might render one snapshot's intersection with another entirely negligible, the implication of which might be a failure to detect. To mitigate this latter threat, we could require that calls be not only present but perhaps in some proportion. The stealthiest of worms might keep their executions brief but recurrent, rendering appropriate response to already deceased processes non-obvious.

However, the strength of our proposed system rests with the power of combinatorics. The more peers in a network, the more likely we are to detect correlations, even in the face of adversarial randomness. We are helped by inherent boundaries in our proxy for behavior: with only finitely many system calls, a worm can only vary so much, whereas networks need not be bounded by only a few hundred peers.

However, the most virulent of worms might attack hosts' ability to take or submit snapshots. After all, disabling software like Wormboy tends not to be difficult, as it is not uncommon for



(a) `sshd.exe`



(b) I-Worm/Mydoom.F

Figure 6: Snapshots from (a) `sshd.exe` and (b) I-Worm/Mydoom.F, using windows of 30 seconds. For each window, the frequency of each service's invocation is depicted as a percentage of the total number of calls into kernel space during that window. For visual clarity, snapshots are unlabeled; distinct shades imply distinct system services. Although `sshd.exe` and I-Worm/Mydoom.F appear similar to Wormboy (because the two invoke a large, common subset of system calls), the two differ in their relative frequencies of invocations.

Windows users to log in with administrative rights (the implication of which is that worms, upon infection, might execute with those same rights). But recent advances by Intel [4] and AMD [1] in virtualization might mitigate this threat by allowing IDSes to operate below worms' radar.

Of course, our proposed architecture's supernodes invite potential denial-of-service attacks, but no more so than other services with any centrality (*e.g.*, DNS). Similarly might worms attack supernodes through submission of bogus or forged snapshots, defense against will likely involve some form of authentication. We will examine in future work these and other network-based threats in more detail.

We will also explore in future work mechanisms for response to worms upon detection.

7. RELATED WORK

In that we generalize the problem of worms' discovery as a problem of detection of widespread and coordinated behavior, our work aligns with research generally focused on anomaly or intrusion detection. Although literature in this space has focused more on Linux, UNIX, and TCP/IP itself than it has on Windows, ideas therein are of particular relevance to our own work.

Somayaji *et al.* [39,40] describe pH, a kernel extension for Linux that monitors processes' execution for unexpected sequences of system calls, though only with respect to a host's own prior behavior. An outgrowth of that research is work by Hofmeyr [15,16], whose Sana Security, Inc. [35] provides "instant protection against a targeted, emerging attack class." Lee *et al.* [20] similarly extend the work of Somayaji *et al.*

Eskin [7] focuses on anomaly detection using learned probability distributions, an approach that we might eventually adopt for more dynamic definitions of snapshots. Of commercial relevance to Wormboy are products from Symantec Corporation [44] and McAfee, Inc. [22], the latter of which offers "zero-day protection against new attacks" by combining behavioral rules with signatures.

Though more network- than host-based, Autograph [19] and Polygraph [25] generate signatures for novel and polymorphic worms, respectively. Singh *et al.* [37] propose methods for automated worm fingerprinting. Ellis *et al.* [6] propose a network application architecture. Jung *et al.* [18] suggest sequential hypothesis testing for scanning worms' detection, while Schechter *et al.* [36] offer improvements on the same. Weaver *et al.* [48] advance cooperative algorithms for worms' containment. Anderson and Li [2] endeavor to separate worm traffic from benign. Williamson [49] proposes throttling viruses, while Twycross and Williamson [46] explore implementation of the same.

Apap *et al.* [3] and Stolfo *et al.* [43] focus on Windows itself, offering algorithms for anomaly detection within the Windows registry. Hu and Mok [17], meanwhile, leverage kernel activity to detect mass-mailing viruses.

8. CONCLUSION

Inherent in automated, behavior-based IDSes for worms is a risk of false positives. We combat this risk with collaboration among peers. In this paper, we vetted this idea using our implementation of Wormboy 2.0, a prototype for host-based, collaborative detection available for other researchers to download and use. We deployed our prototype to a network of 30 hosts running Windows, where we monitored and analyzed 10,776 processes. Using the data gathered from this network, we exposed the utility of temporal consistency (similarity over time in worms' and non-worms' invocations of system calls) in collaborative detection.

We identified properties with which we can distinguish non-worms from worms 99% of the time. We found that a collaborative network, using patterns of system calls and simple heuristics, can detect worms running on multiple hosts. And we found that collaboration among peers significantly reduces the risk of false positives because of the unlikely, simultaneous appearance across peers of non-worm processes with worm-like properties.

In future work, we will expand our deployment and re-evaluate our thresholds for τ, r, r', and ι with additional data. We will question the scalability of our proposed supernodes, consider implications of peers' geography, and explore algorithms for rapid analysis of peers' snapshots. And we will consider how a host-based, collaborative IDS should respond both to worms upon detection and to attacks on its own architecture.

SOURCE CODE

Source code for Wormboy 2.0 is available for download from `http://www.eecs.harvard.edu/~malan/`.

ACKNOWLEDGEMENTS

This work has been funded in part by NSF Trusted Computing grant CCR-0310877 and by gifts from Microsoft. Our thanks to Michael Mitzenmacher of Harvard University and David van Dyk of the University of California at Irvine for their assistance with this work. And our thanks to Wellie Chao, Rei Diaz, Anthony DiComo, Breanne Duncan, Kathleen Durant, Kelly Heffner, Glenn Holloway, Matthew Fluet, Matthew Foroughi, Dianne Malan, Lauren Malan, Tomas Mikuckis, Gordon Murphy, Daniel Peng, Christopher Power, Roman Rubinstein, Andrew Smith, Samantha Smith, Shane Smith, Lifan Yang, and Xin-Xin Zeng for welcoming Wormboy into their kernels.

REFERENCES

[1] Advanced Micro Devices, Inc. AMD's Virtualization Solutions. `enterprise.amd.com/us-en/Solutions/Consolidation/virtualization.aspx`.

[2] E. Anderson and J. Li. Aggregating Detectors for New Worm Identification. In *USENIX 2004 Work-in-Progress Reports*. USENIX, June 2004.

[3] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proc. of the 5th Int'l Symposium on Recent Advances in Intrusion Detection*, 2002.

[4] Intel Corp. Intel Virtualization Technology. `www.intel.com/technology/computing/vptech/`.

[5] P. Dabak, S. Phadke, and M. Borate. *Undocumented Windows NT.* M&T Books, 1999.

[6] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A Behavioral Approach to Worm Wetection. In *Proc. of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, New York, NY, USA, 2004. ACM Press.

[7] E. Eskin. Anomaly Detection over Noisy Data Using Learned Probability Distributions. In *Proc. of the 17th International Conference on Machine Learning*, 2000.

[8] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proc. of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[9] Grisoft Inc. `www.grisoft.com`.

[10] J. Gulbrandsen. How Do Windows NT System Calls REALLY Work? `www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8035/`, August 2004.

[11] J. Gulbrandsen. System Call Optimization with the SYSENTER Instruction. `www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8223/`, October 2004.

[12] J. Harris. YAC: Yet Another Caller ID Program. `sunflowerhead.com/software/yac/`.

[13] B. Henderson. XML-RPC for C and C++. `xmlrpc-c.sourceforge.net`.

[14] N. P. Herath. Adding Services To The NT Kernel. `microsoft.public.win32.programmer.kernel`, October 1998.

[15] S. A. Hofmeyr. *An Immunological Model of Distributed Detection and Its Application to Computer Security.* PhD thesis, 1999.

[16] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[17] R. Hu and A. K. Mok. Detecting Unknown Massive Mailing Viruses Using Proactive Methods. In *Proc. of the 7th Int'l Symposium on Recent Advances in Intrusion Detection*, 2004.

[18] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2004.

[19] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[20] W. Lee, S. J. Stolfo, and P. K. Chan. *Learning Patterns from Unix Process Execution Traces for Intrusion Detection*, pages 50–56. AAAI Press, 1997.

[21] D. J. Malan and M. D. Smith. Host-Based Detection of Worms through Peer-to-Peer Cooperation. In *Proc. of the 2005 ACM Workshop on Rapid Malcode*, New York, NY, USA, 2005. ACM Press.

[22] McAfee, Inc. `www.mcafee.com`.

[23] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[24] G. Nebbett. *Windows NT/2000 Native API Reference.* MTP, 2000.

[25] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures For Polymorphic Worms. In *USENIX Security Symposium*, 2005.

[26] PC World Communications, Inc. WorldBench 5. `www.worldbench.com`.

[27] M. Pietrek. Poking Around Under the Hood: A Programmer's View of Windows NT 4.0. *Microsoft Systems Journal*, August 1996. `www.microsoft.com/msj/archive/s413.aspx`.

[28] The Metasploit Project. Windows System Call Table (NT/2000/XP/2003). `www.metasploit.com/users/opcode/syscalls.html`.

[29] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, pages 257–272, 2003.

[30] T. J. Robbins. Windows NT System Service Table Hooking. `www.wiretapped.net/~fyre/sst.html`.

[31] P. Roberts. Mydoom Sets Speed Records. `www.pcworld.com/news/article/0,aid,114461,00.asp`.

[32] M. Russinovich. Inside the Native API. `www.sysinternals.com/Information/NativeApi.html`, 1998.

[33] T. Sabin. Personal correspondence.

[34] T. Sabin. Strace for NT. `www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm`.

[35] Sana Security, Inc. `www.sanasecurity.com`.

[36] S. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *7th Int'l Symposium on Recent Advances in Intrusion Detection*, French Riviera, France, September 2004.

[37] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *OSDI*, pages 45–60, 2004.

[38] V. Smirnov. Re: Hooking system call from driver. NTDEV – Windows System Software Developers List, April 2002.

[39] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proc. of the 9th USENIX Security Symposium*, August 2000.

[40] A. B. Somayaji. *Operating System Stability and Security through Process Homeostasis.* PhD thesis, 2002.

[41] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proc. of the 2004 ACM Workshop on Rapid Malcode*, pages 33–42, New York, NY, USA, 2004. ACM Press.

[42] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. of the 11th USENIX Security Symposium*, August 2002.

[43] S. J. Stolfo, F. Apap, E. Eskin, K. Heller, S. Hershkop, A. Honig, and K. Svore. *A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection*, volume 13 of *Journal of Computer Security*, pages 659–693. 2005.

[44] Symantec Corporation. `www.symantec.com`.

[45] B. Tucker. SoBig.F breaks virus speed records. `www.cnn.com/2003/TECH/internet/08/21/sobig.virus/`.

[46] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *USENIX Security Symposium*, pages 285–294, 2003.

[47] UserLand Software, Inc. XML-RPC Home Page. `www.xmlrpc.com`.

[48] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *USENIX Security Symposium*, pages 29–44, 2004.

[49] M. M. Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172R1, HP Labs, December 2002.