Host-Based Detection of Worms through Peer-to-Peer Cooperation

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

ABSTRACT

We propose a host-based, runtime defense against worms that achieves negligible risk of false positives through peer-to-peer cooperation. We view correlation among otherwise independent peers' behavior as anomalous behavior, indication of a fast-spreading worm. We detect correlation by exploiting worms' *temporal consistency*, similarity (low temporal variance) in worms' invocations of system calls. We evaluate our ideas on Windows XP with Service Pack 2 using traces of nine variants of worms and twenty-five non-worms, including ten commercial applications and fifteen processes native to the platform. We find that two peers, upon exchanging snapshots of their internal behavior, defined with frequency distributions of system calls, can decide that they are, more likely than not, executing a worm between 76% and 97% of the time. More importantly, we find that the probability that peers might err, judging a non-worm a worm, is negligible.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Measurement, Security

Keywords

native API, P2P, peer-to-peer, system calls, system services, temporal consistency, Win32, Windows, worms

1. INTRODUCTION

The fastest of worms do not allow time for human intervention [16,23,33,34,37]. Necessary is an automated defense, the first step toward which is detection itself. But detection must be both accurate and rapid. Defenses as high in false positives as they are low in overhead are perhaps just as bad as no defenses at all: both put systems' usability at risk.

We propose a host-based, runtime defense against worms that leverages peer-to-peer (P2P) cooperation to lower its risk of false positives. We claim that, by exchanging snapshots of their internal behavior alone (regardless of network traffic), peers can detect actions so correlated in time as to be more likely those of a fast-spreading worm than not.

Common today are defenses based on automated recognizance of signatures, sequences of bytes indicating some worm's presence in memory or network traffic. Such defenses are fast, and specificity of signatures renders false positives unlikely. But the protections are limited: systems are safe from only those worms for which researchers have had time to craft signatures, and signature-based defenses can be defeated by metamorphic or polymorphic worms [13, 36].

Behavior-based defenses, which monitor systems for anomalous (*e.g.*, yet unseen) behavior, are an alternative, perhaps less susceptible to defeat by mere transformations of text, insofar as they judge the effect of code more than they do its appearance. But this resilience comes at a cost: accuracy or rapidity. Faced with some anomalous action, behavior-based defenses must either block that action, potentially impeding desired behavior, or wait for the user's judgement. Such defenses can be defeated by users themselves, annoyed or confounded by too many false positives or prompts.

Through P2P cooperation can we obviate the need for such manual intervention and still lower our risk of false positives. Worms stand out among other processes not so much for their novelty but for their simplicity and periodicity: their design is to spread, their execution thus cyclical. Granted, even the most innocuous of applications can evince cyclical behavior reminiscent of attacking worms. But less likely are we to see such behavior in near lockstep on multiple hosts, unless triggered by some threat.[1] Through P2P cooperation, then, can we lower our risk of false positives by requiring that individual hosts no longer decide a worm's presence but a cooperative instead.

We focus in this paper on precisely this problem of collaborative detection with few false positives. We find that we can detect worms by leveraging collaborative analysis of

[1]To allow for coordinated behavior in distributed applications (*e.g.*, Entropia [2]), it would suffice to maintain white lists. Such applications often provide protections in their virtual machines against ill-behaved and malicious grid programs.

WORM'05, November 11, 2005, Fairfax, Virginia, USA.

peers' runtime behavior while reducing the collective's risk of false positives. Specifically, we find that two peers, upon exchanging snapshots of their internal behavior, can decide that they are, more likely than not, both executing the same worm between 76% and 97% of the time. Moreover, we find that, while certain non-worms can exhibit sufficiently cyclical behavior as to be potentially mistaken by peers for worms themselves, such mistakes can be avoided. Finally, we find that two peers are unlikely to mistake a non-worm executing on one for a worm executing on the other.

In the section that follows, we motivate our inquiry into the viability of collaborative detection by expounding on our proposal for a host-based, runtime defense leveraging P2P cooperation. In Section 3, we describe the methodology with which we explore the matter, and, in Section 4, we present results on the efficacy of our approach. In Section 5, we discuss threats to P2P cooperation. Finally, in Sections 6 and 7, respectively, we explore related work and conclude.

2. TEMPORAL CONSISTENCY

Conventional host- and behavior-based defenses dictate that hosts evaluate some current action vis-à-vis prior actions; a host's behavior is deemed anomalous if it differs from that host's prior behavior. We offer an alternative definition of anomalous behavior. We propose that a host evaluate some current action vis-à-vis its peers' current actions; a host's behavior is deemed anomalous if it correlates all too well with other, otherwise independent, hosts' behavior. We argue that anomalous behavior, induced by some worm, can be detected because of *temporal consistency*, similarity (low temporal variance) in invocations of system calls.

We exploit the reality that worms' behavior tends toward simplicity and periodicity. Of course, non-worms' behavior, on occasion, can resemble that of worms', as we discuss in Section 4. But so relatively few are a worms' actions, more likely are we to detect them in lockstep on multiple peers than those of larger, more complicated applications with more code paths. Through P2P cooperation, then, can we harness the power of combinatorics to lower our risk of false positives.

Specifically, we propose a network of peers (Figure 1), each running software designed to take snapshots of processes' calls into kernel space. On some schedule do peers exchange the relative frequencies of these calls during some window of time (Figure 2). If the cooperative finds too many similarities among snapshots, a worm is assumed present and a response initiated.

Out of this vision comes a number of problems: how best to detect similarities, how best to exchange snapshots, and how best to respond. We address in this work the first of these challenges: the problem of detection. We assume, for the purposes of this inquiry, that communication among peers is not only instantaneous and infinite but also centralized at some node. We bound, through experimentation with both worms and non-worms, the probabilities with which peers might detect and mistake worms. We leave, for subsequent work, relaxation of these assumptions, focusing now on whether peers might detect worms at all with few false positives.

We recognize that execution of some worm within a network of hosts might not be perfectly synchronized, as hosts might not have become infected at the same moment in time. We must therefore tolerate some difference in timing, even

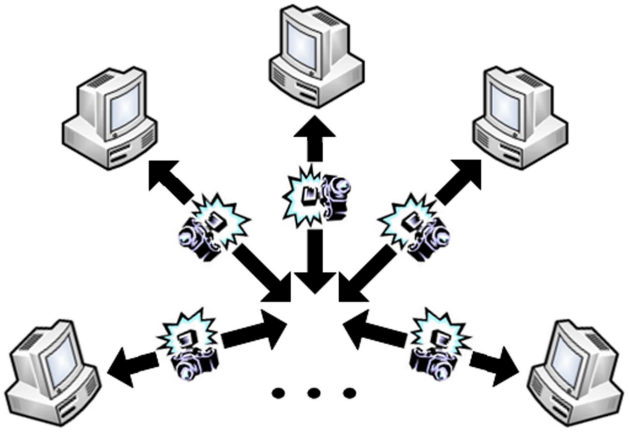

Figure 1: Our vision of collaborative detection of fast-spreading worms using P2P networks. On some schedule, peers exchange snapshots of their internal behavior; too many similarities suggest anomalous behavior, a worm's presence.

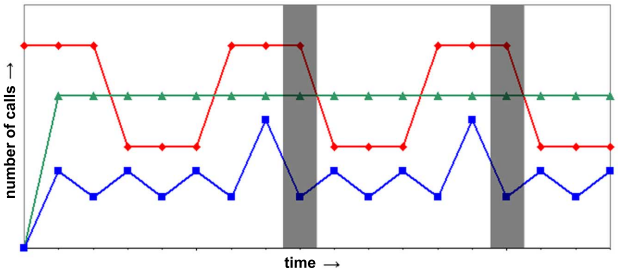

Figure 2: Hypothetical trace of a process's invocation over time of three system calls, each of which is plotted as a separate line. Point (i, j) indicates j invocations of some system call around time i. Shaded are two samples, representative of snapshots that might be exchanged by two peers. The more peers that exchange snapshots similar to these, the more correlated is their behavior, and the more likely are they infected by some worm.

though hosts are to exchange snapshots of their behavior on some schedule.

We offer two effective measures of similarity, both of which are tolerant of offsets in timing. Neither measure expects perfect matches in peers' sequences of system calls, lest it be too sensitive to slight variance in worms' execution and to randomization along code paths by the stealthiest of worms.

2.1 Measuring Similarity with Edit Distance

Our first measure of similarity treats snapshots of hosts' behavior as ordered sets of system calls, enumerated according to their frequencies of execution during some window of time. Each such set is of the form $S = (s_0, s_1, \ldots, s_{n-1})$, where each s_i is a unique token representing some system call, and the relative frequency of s_i within the snapshot is greater than or equal to that of s_j, for $i < j$. A process is said to be temporally consistent if a majority of snapshots of its execution over time are similar.

We judge the similarity of two snapshots by way of the

edit distance between them, which we define here as the number of insertions, deletions, and substitutions required to transform one set of tokens into the other. Inasmuch as this distance, d, is thus bounded by the larger of $|S_1|$ and $|S_2|$, for two snapshots, S_1 and S_2, we define the percentage of similarity between the snapshots as $1 - \frac{d}{\max(|S_1|,|S_2|)}$. If $1 - \frac{d}{\max(|S_1|,|S_2|)} \geq 0.5$ for a majority of pairs of snapshots over time, the process to which those snapshots pertain is said to be temporally consistent.

In that it considers invocations of system calls in the aggregate, this measure finds similarity where mere pattern matching (to which edit distance is conventionally applied) might fail. But it is particularly sensitive to fluctuations in system calls' order (as might be induced by branches in a worm's call graph).

2.2 Measuring Similarity with Intersection

Our second measure of similarity treats snapshots of hosts' behavior as unordered sets of system calls invoked during some window of time. Each such set is again of the form $S = (s_0, s_1, \ldots, s_{n-1})$, where each s_i is a unique token representing some system call, but no ordering is imposed on the set. A process is still said to be temporally consistent if a majority of snapshots of its execution over time are similar.

But we judge the similarity of two snapshots, S_1 and S_2, by way of $S_1 \cap S_2$. We define the percentage of similarity between two snapshots as $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)}$. If $\frac{|S_1 \cap S_2|}{\max(|S_1|,|S_2|)} \geq 0.5$ for a majority of pairs of snapshots over time, the process to which the snapshots pertain is said to be temporally consistent.

Blind as this version is to order, it allows for the emergence of patterns despite slight differences in execution.

3. METHODOLOGY

We target Windows XP with Service Pack 2 (WinXP SP2) for our proposed defense, as the platform offers a richness of available worms (important in any behavioral study) and is a perpetual recipient of innovative attacks. We look, as have others before us [5, 10, 21, 31, 32], albeit on Linux and UNIX, to system calls as a proxy for hosts' behavior. And we deploy our two measures of similarity to quantify the probability that snapshots of calls into kernel space do, in fact, belong to the same executable.

As our approach to detection does not require synchronization among peers, we are able to evaluate our proposal's viability with traces of hosts' behavior; we do not require the experimental overhead of an actual network of peers. With such traces, we simulate snapshots' instantaneous exchange between pairs of peers and compute the probabilities with which those peers can decide that they are, more likely than not, both executing the same worm.

As part of this work, we have implemented Wormboy 1.0, software with which to gather these traces. And we have traced the behavior of WinXP SP2's fastest worms and commonest non-worms. We elaborate here on our choice of system calls (Section 3.1), our implementation of Wormboy (Section 3.2), and our experimentation on WinXP SP2 (Section 3.3). And we frame our inquiry into the efficacy of collaborative detection as a set research questions (Section 3.4).

3.1 A Proxy for Hosts' Behavior

To the extent that they circumscribe kernel space, restricting execution's passage from Ring-3 to Ring-0, system calls enable summarization of code into low-level, but still semantically cogent, building blocks. Other summaries might be useful, particularly bytes received or even sent. But, covered in literature as is the behavior of worms' network traffic already [12, 18, 29], we focus instead for temporal consistency on worms' utilization of WinXP SP2's *native API*, the nearest equivalent of Linux's and UNIX's system calls.

This native API comprises 284 functions, known also as *system services*, 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 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- 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).[2] 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`. [3, 7–9, 20, 24]

3.2 Wormboy 1.0

To capture the behavior of WinXP SP2 with respect to its system services, we have implemented Wormboy 1.0, a kernel-mode driver that inserts hooks into `_KeServiceDescriptorTable` before and after all but two system services.[3,4] Inspired by Strace for NT [26], as well as by work by Nebbett [17] and Dabak *et al.* [3], Wormboy not only captures a call's service ID and input parameters, but also its output parameters and return value, along with a caller's name, process ID, thread ID, and mode. Though Wormboy will ultimately serve as the core of a real-time defense, the driver, for now, captures all such data to disk, timestamping and sequencing each entry per trace, so that we might experiment offline with different approaches to detection. A link to Wormboy's source code is offered at this paper's end.

[2]On older CPUs, `SharedUserData!SystemCallStub` executes a slower `INT 2e` instruction.

[3]By default, `_KeServiceDescriptorTable` is read-only, so Wormboy first disables the `WP` bit in register `CR0` [22, 30]. Alternatively, protection of kernel memory itself could be relaxed, albeit dangerously, by creating registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\EnforceWriteProtection` with a `DWORD` value of `0x0` [26].

[4]WinXP SP2 appears to make certain assumptions about system services `NtContinue` and `NtRaiseException`, whereby it attempts to manipulate the stack frame based on register `EBP` [25]; inasmuch as our hooks insert a frame of their own, we do not hook these services to avoid system crashes.

Worms	Non-Worms	
	Commercial Applications	**WinXP SP2 Processes**
I-Worm/Bagle.Q	Adobe Photoshop 7.0.1	`alg.exe`
I-Worm/Bagle.S	Microsoft Access XP SP2	`csrss.exe`
I-Worm/Jobaka.A	Microsoft Excel XP SP2	`defrag.exe`
I-Worm/Mydoom.D	Microsoft Outlook XP SP2	`dfrgntfs.exe`
I-Worm/Mydoom.F	Microsoft Powerpoint XP SP2	`explorer.exe`
I-Worm/Sasser.B	Microsoft Word XP SP2	`helpsvc.exe`
I-Worm/Sasser.D	Network Benchmark Client 1.0.3	`lsass.exe`
Worm/Lovesan.A	Nullsoft Winamp 5.094	`msmsgs.exe`
Worm/Lovesan.H	Windows Media Encoder 9.0	`services.exe`
	WinZip 8.1	`spoolsv.exe`
		`svchost.exe`
		`wmiprvse.exe`
		`winlogon.exe`
		`wscntfy.exe`
		`wuauclt.exe`

Table 1: Worms and non-worms whose traces we analyzed.

3.3 Traces of Worms and Non-Worms

To validate our claim of collaborative detection's efficacy, we look to some of WinXP SP2's fastest worms and commonest non-worms to date. We base our results, put forth in Section 4, on traces of nine variants of worms and twenty-five non-worms, including ten commercial applications and fifteen processes native to WinXP SP2 (Table 1).[5]

For each worm, we have traced its live activity for fifteen minutes, more than enough for its recurrent behavior to surface. For each commercial application save one, we have traced its execution under PC Magazine's WebBench 5.0 [14] or PC World's WorldBench 5 [19] benchmarking suites.[6] For each native process, we have traced its execution during twenty-four hours of user-free intervention.

Of course, none of these traces, save those of the worms, may be representative of normal activity, if such can even be said to exist. But, insofar as these traces have been gathered in environments as deterministic as possible, we argue that they actually allow us to estimate lower bounds on peers' ability to detect or mistake worms; it's hard to imagine programs more cyclical (and thus worm-like) than those executing repeated tests or taking no input.

3.4 Research Questions

To detect novel worms by leveraging collaborative analysis of peers' runtime behavior, we must demonstrate that worms tend to stand out in traces of system behavior based on calls to system services. Given two or more samples from those very same traces (*i.e.*, snapshots of behavior), distinguishing an attacking worm from an otherwise benevolent application reduces to the following three questions, each irrespective of our timing of samples.

1. *How likely is a worm to look like itself?* The more similar a worm's execution during some window of time to its execution during any other, the more capable should peers be to correlate actions. Moreover, the more similar a worm with respect to itself, the less it should matter when peers sample their behavior. We thus inquire as to whether worms are temporally consistent.

2. *How likely is a non-worm to look like itself?* The more similar a non-worm's execution during some window of time to its execution during any other, the more likely might peers be to think it a worm. We thus inquire as to whether non-worms are temporally consistent.

3. *How likely is a non-worm to look like a worm?* The more similar a non-worm's execution to that of a worm, the more likely might peers be to mistake the benign for the malevolent. We thus inquire as to whether worms manifest similarities with non-worms.

4. RESULTS

We present in this section our results for the research questions of Section 3.4.

4.1 How likely is a worm to look like itself?

A worm is remarkably likely to look like itself, though it depends on the measure of similarity. We find that, while edit distance allows us to notice with near certainty (at least 95%) the similarity, with respect to themselves, of I-Worm/Sasser.D, Worm/Lovesan.A, and Worm/Lovesan.H, using a window size of 15 seconds, the metric proves less effective on other variants (Table 2), even for windows as wide as 30 seconds. Bagle's variants, in particular, appear resistant to classification as temporally consistent using the metric, with no more than 14% of possible pairs of snapshots resembling each other. The disparity, though significant, is not surprising, if we consider the traces themselves.

[5]We call the worms by their names according to AVG Free Edition 7.0.322 [6].

[6]We traced Nullsoft Winamp 5.094 as it played an MP3 of James Horner's 19-minute "Titanic Suite," encoded at 160 kbps.

	5	15	30
I-Worm/Bagle.Q	14%	11%	10%
I-Worm/Bagle.S	14%	11%	11%
I-Worm/Jobaka.A	59%	50%	69%
I-Worm/Mydoom.D	92%	81%	73%
I-Worm/Mydoom.F	17%	31%	41%
I-Worm/Sasser.B	60%	54%	72%
I-Worm/Sasser.D	95%	97%	93%
Worm/Lovesan.A	99%	98%	93%
Worm/Lovesan.H	47%	95%	93%

Table 2: Probability with which two peers, upon exchanging snapshots of their internal behavior, can decide using *edit distance* alone that they are, more likely than not, both executing the same worm during some window of time, for window sizes of 5, 15, and 30 seconds. In other words, percentages of all possible pairs of samples from some worm for which $1 - \frac{d}{\max(|S_1|,|S_2|)} \geq 0.5$, where S_1 and S_2 are snapshots, treated as ordered sets, and d is the edit distance between them.

	5	15	30
I-Worm/Bagle.Q	80%	76%	81%
I-Worm/Bagle.S	82%	76%	73%
I-Worm/Jobaka.A	99%	97%	93%
I-Worm/Mydoom.D	99%	97%	93%
I-Worm/Mydoom.F	99%	97%	93%
I-Worm/Sasser.B	99%	97%	93%
I-Worm/Sasser.D	99%	97%	93%
Worm/Lovesan.A	99%	97%	93%
Worm/Lovesan.H	49%	97%	93%

Table 3: Probability with which two peers, upon exchanging snapshots of their internal behavior, can decide using *intersection* alone that they are, more likely than not, both executing the same worm during some window of time, for window sizes of 5, 15, and 30 seconds. 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, treated as unordered sets.

For instance, whereas Worm/Lovesan.A (Figure 3) manifests an obvious, nearly constant, pattern, I-Worm/Bagle.Q (Figure 4) boasts a less obvious pattern, clouded by overlapping frequencies.

With less precise measures, though, we can filter such noise. If we consider only calls' intersection but not relative frequencies, we notice more trends. We now notice with near certainty (97%), using a window size of 15 seconds, every one of our worms save Bagle; but now even Bagle appears temporally consistent (Table 3).

Still worthy of note, though not unexpected, is Worm/Lovesan.H, which resists detection, no matter our metric, using a window size of 5 seconds. Such narrow windows simply fail to capture this worm's periodicity (Figure 5); wider windows do capture its periodicity (Figure 6).

4.2 How likely is a non-worm to look like itself?

A non-worm is not nearly as likely to resemble itself as is a worm to resemble itself. Of all our non-worms examined, only Nullsoft Winamp and `alg.exe` boasted traces for which more than 90% of 15-second snapshots resembled each other, no matter the metric. And only `alg.exe` boasted a trace for which more than 90% of 30-second snapshots resembled each other, no matter the metric.

But `alg.exe`, during our twenty-four-hour run, made only 2295 calls to system services, an average of no more than one per second. By contrast, even our "slowest" of worms, I-Worm/Jobaka.A, averaged sixty-four such calls per second. Insofar as processes averaging nearly zero calls per second do not likely belong to fast-spreading worms, we simply require for temporal consistency that snapshots not be so empty.

Nullsoft Winamp, by contrast, averaged 896 calls to system services per second, so its temporal consistency necessitates more intelligent filtration. To discourage false positives, whereby we classify non-worms as worms, we propose to ignore binaries for which we have ample history or read-only hashes (as we might for an application like Nullsoft Winamp, if installed and executed with consent), against which we might compare processes executing in memory. If a process's behavior or text is as we expect, it is not likely a worm. Worms are by nature, after all, binaries foreign to a machine, suddenly installed without users' consent, for which we are unlikely to have history or hashes. In future work will we assess this filter's efficacy.

It is in this analysis of non-worms that the sensitivities of our measures of similarity become apparent. If we lower our threshold for detection, requiring only that 50% (and not 90%) of snapshots resemble each other, we find that edit distance deems not only Nullsoft Winamp and `alg.exe` worms but seven other binaries as well. If we turn instead to intersection, as we did to catch Bagle, we realize that this metric's power comes at a cost: eleven binaries besides Nullsoft Winamp and `alg.exe` are deemed worms. However, a higher threshold (90%) does avoid these false positives.

4.3 How likely is a non-worm to look like a worm?

Through exhaustive comparison of every possible snapshot from each worm against every possible snapshot from each non-worm, we find that only one non-worm's behavior resembles, more often than not, that of a worm: Network Benchmark Client is similar to I-Worm/Jobaka.A, I-Worm/Sasser.B, and I-Worm/Sasser.D, if intersection is our metric. But the resemblance is neither surprising nor troubling, as Network Benchmark Client is practically a worm itself, designed to fork five threads, each of which induces stress on a server by initiating TCP sockets in rapid succession.

5. THREATS TO P2P COOPERATION

Threats to our proposed scheme for collaborative detection include worms designed not to exhibit similarity in their invocations of system services, no matter our measure. Ran-

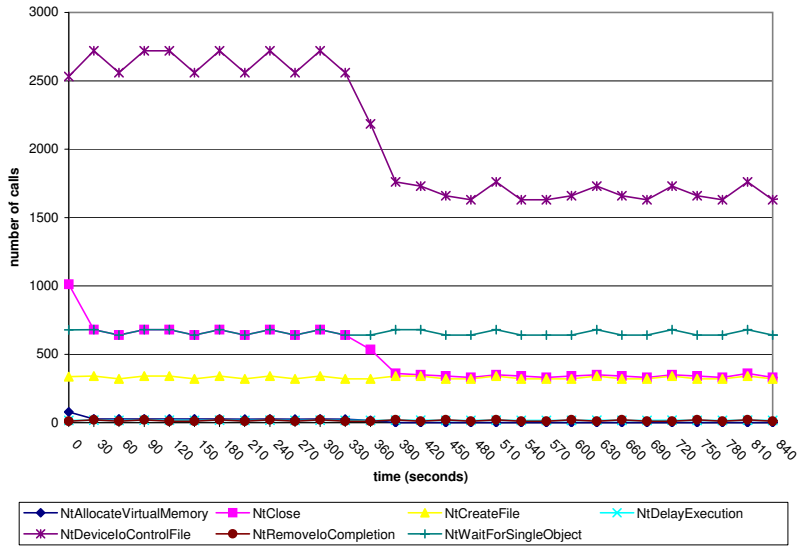

Figure 3: Calls to system services by Worm/Lovesan.A per 30-second window of time. Point (i, j) indicates j calls to some service between times i and $i + 30$. Both edit distance and intersection capture this worm's pattern of activity.

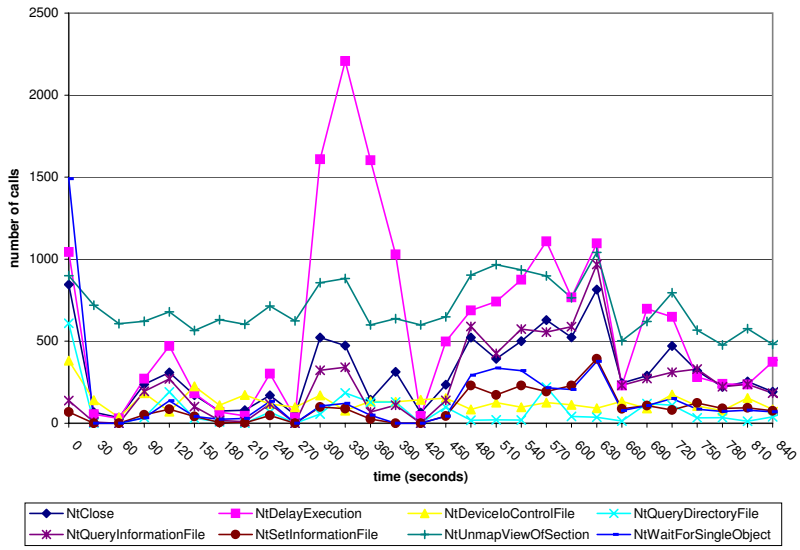

Figure 4: Calls to system services by I-Worm/Bagle.Q per 30-second window of time. Point (i, j) indicates j calls to some service between times i and $i + 30$. For visual clarity, less frequently called system services are not pictured. Edit distance fails to capture this worm's pattern of activity because of overlapping frequencies; intersection does capture the pattern.

dom calls to system services by the stealthiest of worms could skew analysis of peers' behavior, particularly for our measure of similarity based on edit distance, insofar as the metric is sensitive to changes in order. However, to mitigate this threat, we could consider order but allow for transpositions, requiring only that the bubble-sort distance between two snapshots (the number of swaps that bubble sort would make to transform one ordered set into the other) be within some bound. Our measure based on intersection is similarly vulnerable to adversarial randomness, as the stealthiest of worms might, on occasion, invoke all possible services in series, simply to render any intersection with another peer's snapshot negligible, thereby masking its presence on some host. To mitigate this threat, though, we could simply require that calls be present not necessarily in some order but at least in some proportion.

Of course, the more peers in a network, the more likely we are to discover correlations, even in the face of randomness. There are only so many ways that fast-spreading worms might achieve malicious effects *rapidly*, bounded by time as they are by their very definition.

Just as future work will address how best to exchange snapshots and how best to respond to worms, once detected, it will also address additional threats, involving not only adversarial randomness but also matters of authentication, availability, efficiency, and integrity.

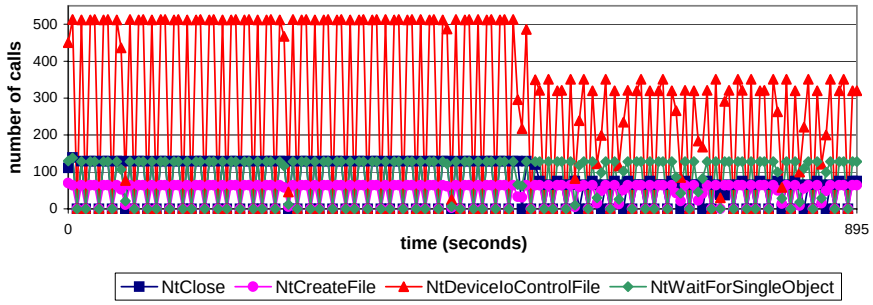

Figure 5: Calls to system services by Worm/Lovesan.H per 5-second window of time. Point (i, j) indicates j calls to some service between times i and $i + 5$. For visual clarity, less frequently called system services are not pictured; similarly are most x-axis labels omitted. 5-second windows are not adequate to capture periodicity in this worm's behavior.

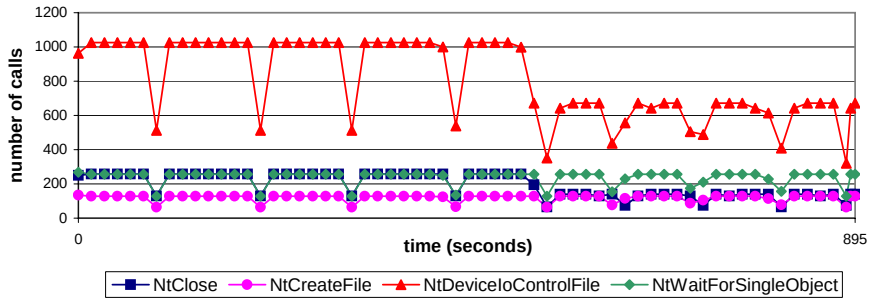

Figure 6: Calls to system services by Worm/Lovesan.H per 15-second window of time. Point (i, j) indicates j calls to some service between times i and $i+15$. For visual clarity, less frequently called system services are not pictured; similarly are most x-axis labels omitted. 15-second windows are adequate to capture periodicity in this worm's behavior.

6. RELATED WORK

Insofar as our own work aspires to generalize the problem of worms' discovery away from recognizance of pre-defined signatures toward detection of widespread and coordinated behavior, it falls within an area of research more generally focused on anomaly or intrusion detection, be it network- or host-based. Though a dearth of published work exists for Win32, a growing body of literature exists for Linux, UNIX, and TCP/IP alike. Of relevance to our own work is that of Somayaji *et al.* [31,32], whose Linux-based pH monitors processes' execution for unexpected sequences of system calls, though only with respect to a host's own prior behavior. An outgrowth of the same is work by Hofmeyr [10, 11], whose Sana Security, Inc. [27] provides "instant protection against a targeted, emerging attack class." Similar are products from Symantec Corporation [35] and McAfee, Inc. [15], the latter of which offers "zero-day protection against new attacks" by combining behavioral rules with signatures.

Though more network- than host-based, worthy of note are Autograph [12] and Polygraph [18], signature-generation systems for novel and polymorphic worms, respectively. Also of interest are the methods for automated worm fingerprinting of Singh *et al.* [29] as well the network application architecture of Ellis *et al.* [4]. Jung *et al.*, meanwhile, propose sequential hypothesis testing for scanning worms' detection, while Schechter *et al.* [28] offer improvements on the same. Twycross and Williamson [38] propose that worms be throttled: instead of preventing such programs from entering a system, they seek to prevent them from leaving. Weaver *et al.* [39] similarly advance cooperative algorithms for worms' containment. In progress is work by Anderson and Li [1] on separating worm traffic from benign.

7. CONCLUSION

Host-based detection of worms through P2P cooperation is possible with negligible risk of false positives, as we demonstrate through analysis on WinXP SP2 of nine variants of worms and twenty-five non-worms. Our result follows from a definition of anomalous behavior as correlation among otherwise independent peers' behavior. For the set of worms and non-worms tested, we find that two peers, upon exchanging snapshots of their internal behavior, defined in terms of frequency distributions of calls to system services, can detect execution of some worm between 76% and 97% of the time because of worms' temporal consistency. More significantly, the probability of false positives is negligible, and, in those rare cases in which non-worms manifest temporal consistency, simple filters eliminate the false positives.

This paper focuses entirely on the problem of collaborative detection of worms. We will relax in subsequent work our assumptions that communication among peers is instantaneous, the communication bandwidth infinite, and the detection centralized. We will also explore what constitutes an appropriate response for such a cooperative defense.

ACKNOWLEDGEMENTS

This work has been funded in part by NSF Trusted Computing grant CCR-0310877 and by gifts from Microsoft. Many thanks to Michael Mitzenmacher and Glenn Holloway of Harvard University and to David van Dyk of the University of California at Irvine for their assistance with this work.

SOURCE CODE

Source code for Wormboy 1.0 is available for download from `http://www.eecs.harvard.edu/~malan/`.

REFERENCES

[1] Eric Anderson and Jun Li. Aggregating Detectors for New Worm Identification. In *USENIX 2004 Work-in-Progress Reports*. USENIX, June 2004.

[2] B. Calder, A. Chien, J. Wang, and D. Yang. The Entropia Virtual Machine for Desktop Grids. In *Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)*, pages 186–196, Chicago, IL, June 2005.

[3] Prasad Dabak, Sandeep Phadke, and Milind Borate. *Undocumented Windows NT*. M&T Books, 1999.

[4] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A Behavioral Approach to Worm Wetection. In *WORM '04: Proceedings of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, New York, NY, USA, 2004. ACM Press.

[5] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[6] Grisoft Inc. `http://www.grisoft.com/`.

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

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

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

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

[11] Steven Andrew Hofmeyr. *An Immunological Model of Distributed Detection and Its Application to Computer Security*. PhD thesis, 1999.

[12] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[13] Oleg Kolesnikov and Wenke Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical Report GIT-CC-05-09, Georgia Institute of Technology, 2005.

[14] PC Magazine. WebBench 5.0. `http://www.pcmag.com/benchmarks/`.

[15] McAfee, Inc. `http://www.mcafee.com/`.

[16] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[17] Gary Nebbett. *Windows NT/2000 Native API Reference*. MTP, 2000.

[18] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures For Polymorphic Worms. In *USENIX Security Symposium*, 2005.

[19] PC World Communications, Inc. WorldBench 5. `http://www.worldbench.com/`.

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

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

[22] Tim J. Robbins. Windows NT System Service Table Hooking. `http://www.wiretapped.net/~fyre/sst.html`.

[23] Paul Roberts. Mydoom Sets Speed Records. `http://www.pcworld.com/news/article/0,aid,114461,00.asp`.

[24] Mark Russinovich. Inside the Native API. `http://www.sysinternals.com/Information/NativeApi.html`, 1998.

[25] Todd Sabin. Personal correspondence.

[26] Todd Sabin. Strace for NT. `http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm`.

[27] Sana Security, Inc. `http://www.sanasecurity.com/`.

[28] Stuart Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast Detection of Scanning Worm Infections. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, French Riviera, France, September 2004.

[29] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *OSDI*, pages 45–60, 2004.

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

[31] Anil Somayaji and Stephanie Forrest. Automated Response Using System-Call Delays. In *Proceedings of 9th Usenix Security Symposium*, August 2000.

[32] Anil Buntwal Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, 2002.

[33] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *WORM '04: Proceedings of the 2004 ACM Workshop on Rapid Malcode*, pages 33–42, New York, NY, USA, 2004. ACM Press.

[34] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[35] Symantec Corporation. `http://www.symantec.com/`.

[36] Péter Ször and Peter Ferrie. Hunting for Metamorphic. In *Proceedings of Virus Bulletin Conference*, pages 123 – 144, September 2001.

[37] Bill Tucker. SoBig.F breaks virus speed records. `http://www.cnn.com/2003/TECH/internet/08/21/sobig.virus/`.

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

[39] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very Fast Containment of Scanning Worms. In *USENIX Security Symposium*, pages 29–44, 2004.