Containment of Network Worms via Per-Process Rate-Limiting

Yuanyuan Zeng Xin Hu Haixiong Wang Kang G. Shin
The University of Michigan
Ann Arbor, Michigan
{gracez,huxin,wanghx,kgshin}@umich.edu

Abhijit Bose
IBM T.J. Watson Research
Hawthorne, New York
bosea@us.ibm.com

ABSTRACT

Network worms pose a serious threat to the Internet infrastructure as well as end-users. Various techniques have been proposed for detection of, and response against worms. A frequently-used and automated response mechanism is to rate-limit outbound worm traffic while maintaining the operation of legitimate applications, offering a gentler alternative to the usual detect-and-block approach. However, most rate-limiting schemes to date only focus on host-level network activities and impose a single threshold on the *entire* host, failing to (i) accommodate network-intensive applications and (ii) effectively contain network worms at the same time. To alleviate these limitations, we propose a per-process-based containment framework in each host that monitors the fine-grained runtime behavior of each process and accordingly assigns the process a suspicion level generated by a machine-learning algorithm. We have also developed a heuristic to optimally map each suspicion level to the rate-limiting threshold. The framework is shown to be effective in containing network worms and allowing the traffic of legitimate programs, achieving lower false-alarm rates.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Design, Security

Keywords

Worm Containment, Rate-Limiting, Behavior Analysis

1. INTRODUCTION

In recent years, there has been an exponential surge in both the number of network worms and the severity of damage they have inflicted [1]. Fast-spreading worms, such as Blaster (2003), MyDoom (2004), Zotob (2005), propagate at an unprecedented rate and can affect most vulnerable systems within tens of minutes. The intent of a worm has evolved from simply replicating itself to installing spyware or backdoors in the victim systems for collecting confidential information and perpetrating other attacks. For example, worms are ideal tools for recruiting compromised machines and constructing large botnets among them, which are then used for mounting other serious attacks, such as spamming and DDoS. Hence, there is a pressing need for efficient mitigation of worm spreading.

To combat fast-spreading worms, numerous solutions have been proposed to detect and automatically respond to worm outbreaks. A widely-used approach is the signature-based detection, which looks for specific signatures (usually raw byte sequences) in the application executables. The disadvantage of this scheme is that it can only detect previously-known worms and can be evaded even with simple variations thereof. Behavior-based detection has recently received considerable attention due to its capability of identifying new attacks [2, 3, 4, 5, 6, 7]. Most of prior work requires direct analysis of the binaries [2, 3] or system call sequences [4, 5, 6]. Also, the purpose of behavior-based detection is to classify each application as malicious or benign, which may result in high false-alarm rates due to the ambiguity of behavior-matching.

For fast and effective containment of worms, an automatic response is of particular interest because any method that requires human intervention is much slower than the spreading speed of current worms. The detect-and-block approaches could eliminate the human intervention in the loop, but they either fail to respond to unknown worms or suffer a high false-positive rate. Rate-limiting is a gentler alternative to the above approaches, and its main idea is to block the propagation of worms while allowing legitimate traffic to go through, by differentiating traffic patterns between legitimate applications and network worms. Rate-limiting cannot completely block worms, but can significantly slow down the propagation of (especially new) worms, allowing for other countermeasures to kick in.

A key to worm containment is the selection of a metric based on which the traffic is rate-limited. Previous research results [8, 9, 10] suggest several metrics derived from host-level network activities, such as distinct IP connection ratio, failed connection ratio, the number of connections without DNS queries, etc. However, they rely solely on host-level network activities and hence are likely to suffer from high false-positive and false-negative rates. False-positives stem from the coarse-grained rate-limiting policies applied indiscriminately to both normal and malicious processes. Legitimate traffic will therefore be affected significantly during a worm outbreak. False-negatives may result from the evasion of detection by worms that have traffic pattern similar to that of normal applications. This prompted us to seek more comprehensive in-

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.
SecureComm'08 September 22 - 25, Istanbul, Turkey
Copyright 2008 ACM 978-1-60558-241-2/08/09 ...$5.00.

formation than just network activities at a fine-grained level, i.e., *per-process* behavior. Another key aspect of rate-limiting is the use of a threshold beyond which the outgoing traffic is blocked. Most existing containment schemes impose a single threshold rate on the *entire* host, such as several distinct IP connections per second, regardless of the application demands, thus affecting the performance of legitimate applications. Sekar *et al.* [11] proposed use of different detection thresholds during different time windows, but they still applied these thresholds without differentiating processes on the host.

We propose a per-process-based containment framework. We define behavior at a higher level than others: a sequence of events rather than that of system calls or API calls. Moreover, the behavior profiles of both worms and normal programs are characterized to utilize the notion of anomaly as well as misuse analysis. Our framework considers not only network activities but also a variety of notable behaviors common to network worms, such as creating AutoRun Registry key, overwriting system directories, etc. To compensate for the inaccuracy of behavior analysis and make the best of behavioral information, we use a machine-learning algorithm, instead of making a clear-cut (binary) decision of malice or innocence, to assign a *suspicion level* to each process based on the comprehensive analysis of its behavior. The suspicion level is then transformed into a threshold to rate-limit the process. Since the generation of a suspicion level incorporates many more process-related properties than network activities alone, the containment scheme can make an accurate and flexible decision on how to rate-limit a process, thus lowering false-positive and false-negative rates.

Our contributions are three-fold. First, we propose a framework incorporating both behavior analysis and containment for automatic defense against fast-spreading network worms. Our framework differs from others in that, instead of per-host rate-limiting based solely on network activities, it incorporates a comprehensive analysis of processes' behavior and performs customized rate-limiting on each process. This fine-grained monitoring and analysis significantly improve the effectiveness of rate-limiting. Second, we apply a machine-learning classification algorithm to generate a suspicion level for each process and develop a heuristic to find an optimal function that maps each suspicion level to a threshold for rate-limiting. Third, we conduct in-depth analysis and simulation using the traces of real-world worm samples plus their variants and normal programs. Our evaluation results show that the proposed scheme can easily accommodate legitimate applications while effectively containing the propagation of network worms. Our fine-grained per-process thresholding can achieve much lower false-positive and false-negative rates than per-host approach.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 provides an overview of our system architecture. Section 4 details the process-level behavior analysis. Section 5 presents the principles of containment. Implementation and evaluation results are presented in Section 6. Section 7 discusses the limitations of our work and their solutions. The paper concludes with Section 8.

2. RELATED WORK

The most relevant to our work is behavior-based detection and rate-limiting. There have been a number of behavior-based approaches, but most of them aim to provide a clear-cut, binary detection result. By contrast, our goal is not to detect and block with binary decisions since false alarms are inevitable in this case. Instead, we first derive a suspicion level or malice probability for each process based on its behavior analysis, and then respond with a process-specific containment scheme.

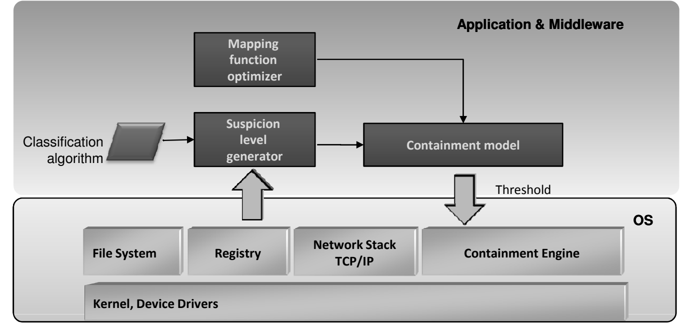

Figure 1: System architecture

Some of previous behavior-analysis approaches rely on examination of executables, such as [2] and [3]. Semantics-aware detection [2] tries to characterize different variations of worms by looking for semantically equivalent instructions in malware variants. In [3], a static analysis is used to identify particular system calls or Internet Explorer API calls that are predefined as malicious. In terms of constructing behavior features, observing system call sequences to identify anomalies is a common approach. Early behavior-based approaches [4, 5, 6] focus on profiling the normal behavior by system call sequences. Our approach differs from them in that behavior is defined at a higher level (i.e., sequence of events rather than sequence of instructions or system calls), and our analysis is conducted at runtime. Moreover, we constructed behavior profiles for both normal and malicious processes.

Most intrusion detection schemes focus on the characteristics of network traffic, such as those in [12, 13]. NetSpy [7] is a tool to automatically discover possibly malicious network activities generated by spyware instances and to generate network-level signatures for them. It collects traffic from an altered machine and then compares it against reference network statistics from clean machines using differential analysis. Our approach differs from [7, 12, 13] in that we consider not only network activities but also file and registry behavior at a process level, rather than on an entire host basis. Our work is similar to behavior-based classification [14] that proposes an automated classification method based on a distance measure. However, the latter focuses on clustering malware into different families, while we are only interested in differentiating normal from malicious programs. Moreover, we use the support vector machine (SVM), a supervised learning approach, in making the best of existing normal and malicious program information, while clustering is a common unsupervised learning procedure.

Several rate-limiting techniques have also been proposed in the literature. The authors of [15] provide a detailed comparison between different methods. Williamson [8] proposes an IP-throttling scheme that restricts the host-level contact rates to unique IPs. Chen *et al.* [9] and Schechter [10] both use the rate of failed connection attempts as the criterion to rate-limit a host. Schechter's approach improves the efficiency by exclusively rate-limiting first-contact connections. Whyte *et al.* [16] utilize the DNS statistics to rate-limit outgoing connections without prior DNS translation. All of these schemes are per-host containment and most of them impose a single threshold for rate-limiting both legitimate and malicious traffic. Our approach monitors fine-grained application behavior and applies rate-limiting on a per-process basis. Each process has a customized threshold so that a higher accuracy in terms of false alarms can be achieved.

3. SYSTEM ARCHITECTURE

Our framework (Figure1) primarily consists of two building blocks:

behavior analysis and containment. The behavior analysis component includes several system monitors and a suspicion-level generator. Runtime behavior for each process is monitored at the OS level, such as Registry, file system and network stack. Process correlation is tracked as well. The suspicion-level generator assigns a suspicion level to each running process by applying the SVM algorithm based on the analysis of its system-wide activities. The suspicion level links process-level behavior to containment. For containment, the mapping function optimizer generates the most appropriate function of transforming the suspicion level to a containment threshold. Both the suspicion-levels and the mapping function are taken as the input to the containment model which then outputs a customized threshold for each process. In what follows, we will detail each component.

4. BEHAVIOR ANALYSIS

The first step to combat the propagation of worms is to identify processes conducting malicious activities in a host system. Previous containment techniques confine themselves to network activities, such as high failure rate and the absence of DNS query, in order to identify suspicious traffic. In this paper, we employ behavior-based analysis that focuses on application run-time behavior including Registry, file system and network. By studying contemporary worms' behaviors, we have observed that they do share certain behavior patterns (e.g., creating autorun registry key, scanning random host IPs) that are different from normal applications.

4.1 Behavior Signature Specification

We define a *behavior signature* as the description of an application's activities in terms of its resource access patterns. Our goal is to develop a simple yet efficient representation of application behavior that maximally differentiates legitimate applications from worms so that suspicion-level information can be generated to facilitate per-process containment. Note that a single activity—such as network access, a file read or written during a worm's life time—alone may appear harmless, while the combination of these activities may reveal a malicious intent. We thus specify a behavior signature as the aggregation of *suspicious* activities that can potentially be exploited by network worms. To design an efficient specification of worm activities, we need to extract the "common" behavior of network worms, which can be understood better by looking at a few notable examples. From real-world worms and their variants, we found that the worm actions can be grouped into 3 categories, taking place at Registry, file system and network stack, where our behavior monitors are deployed.

Registry : A common target of worms is the AutoRun Key `HKLM\Software \Microsoft\ Windows \CurrentVersion\Run`. Most, if not all, worms will add an entry under this key to automatically run themselves when Windows starts up. Examples include Zotob, Win32-Blaster, and W32-Bozori. Some worms also create Registry keys such as `HKEY_CLASSES_ROOT\CLSID\ Random_CLSID\ InprocServer32\(Default)` to conceal its backdoor by injecting into other processes.

File System : Once a system is infected, a worm always downloads its payload from the network to the local file system so that it can be activated again when the system reboots. Almost all worms choose the system directory (e.g., `C:\WINDOWS`) as an ideal place to drop themselves, because normal users seldom inspect the system directory and the worm payload is less noticeable among thousands of system files. Based on this observation, we closely monitor the create and write accesses in system directories.

Network : This category of actions are taken by self-propagating worms, whose goal is to infect as many hosts as possible.

Table 1: Index of behavior vectors

Index	Signature Description
0	Number of First-Contact Connections
1	DNS-to-Connectoin Ratio
2	Number of Suspicious Ports
3	Average Packet Length
4	Number of Packets
5	Modify Dll in System Dir
6	Modify EXE in System Dir
7	Modify other files in System Dir
8	Create Dll to System Dir
9	Create EXE to System Dir
10	Create other files to System Dir
11	Create AutoRun key in Registry
12	Set AutoRun key Value in Registry
13	Create DLL injection key in Registry
14	Set DLL injection key Value in Registry

Note that none of the above activities is inherently malicious, because they are also performed frequently by many normal applications. However, the combination and accumulation of these activities are essential to the detection of malicious intents with a high degree of confidence, as very few legitimate applications will conduct these activities altogether and intensively. We thus construct a vector of behavior features for each process. we also consider process correlation to defend against sophisticated worms that create multiple processes upon execution, which we will describe later. Each feature in the behavior vector represents one type of application behavior of interest. Table 1 summarizes these behavior features that constitute the signature vector. In addition to the behaviors that fall directly into the above three categories, we also take advantage of some auxiliary network features that differentiate worm activities from normal ones. These include DNS-to-connection ratio, suspicious port, and average packet length. The DNS-to-connection ratio is of interest to us because most worms scan *random* IP addresses without DNS queries. The average packet length also provides a hint for suspicious behavior, as worms usually send many identical short packets for both efficiency and fast propagation. Number of suspicious ports records the number of connections initiated by the process to a set of potential vulnerable ports such as 135 and 445. Each behavior feature is associated with a numeric value indicating the number of occurrences of that behavior. The high-level behavior signature for a process is constructed as a vector of all the features, which is then used to determine the suspicion level of the process.

4.2 Suspicion-Level Generator

To respond quickly to fast-spreading (especially previously unknown) worms, we build our behavior analysis upon the *Support Vector Machine* (SVM) [17, 18] that learns the behavior models from both normal and malicious behavior signatures. We collect behaviors from normal applications and worms and generate the corresponding behavior vectors as training data. The SVM algorithm maps training data into a higher-dimensional feature space using a kernel function and determines the maximal margin hyperplane to best separate normal data from malicious data. The hyperplane corresponds to the classification rule. Given a test sample, the SVM calculates the distance of the sample from the separating hyperplane with a sign indicating which class (malicious or benign) the test sample belongs to.

Previous research focused on the binary (malicious or benign) classification and the results are likely to be inaccurate because of the learning procedure. To make the best of the learning model, we calibrate the distance score to a posterior classification probability, which determines how likely a test example belongs to a certain class [19]. The posterior probability is then directly translated into the *suspicion level* between 0 and 1 where 0 (1) means benign (malicious). Apparently, the higher the suspicion level, the more likely the process is malicious, and thus, a stricter containment action should apply. The extension from binary classification to a suspicion level facilitates customization of the containment method for each process. Worm traffic is more likely to be strictly rate-limited while legitimate applications will experience a minor traffic-limiting impact.

It is important to note that our suspicion-level generation is not a one-time rating but a periodic check. It can capture all of runtime behaviors of interest and provide a suspicion level for each process during every time window. Thus, a worm that replaces its process ID with a normal program or attaches itself to a normal program is unlikely to affect our decision. For example, if the Internet Explorer (IE) is the target of the worm, its suspicion level will be high as long as it exhibits some bad behaviors, and its traffic will thus be contained. Some legitimate traffic from IE may be affected, but the process-level containment is the finest-grain one can achieve.

4.3 Process Correlation

Most worms to-date behave badly on their own, while some sophisticated worms may have multiple processes collaboratively conduct malicious activities. To defend against such a worm whose single processes are not malicious enough to trigger effective rate-limiting, we account for process correlation while building the behavior vectors. We track the inter-process relationships and aggregate the behaviors from correlated processes. The behavior vectors are the same for correlated processes such as a parent process and its children. Accordingly, the whole group of correlated processes is assigned the same suspicion level. By maintaining a white list, we can easily exclude some normal processes with correlation such as services and svchost. Thus, it is not difficult to identify a group of processes behaving maliciously altogether.

4.4 Behavior Accumulation

Since the suspicion level is generated every time window, a worm can hide itself by appearing benign for each window but malicious overall. In other words, it may spread suspicious behavior over different time windows or reduce the intensity of malicious activities within a single window, thus decreasing its suspicion level. To deal with such worms, we selectively accumulate the value in each field of the behavior vector. The behavior features worth accumulation are those seldom seen from normal programs, such as creating an autorun key in the Registry or dropping a dll into the system directory, etc. As for some behavior shared by both normal and malicious programs such as outgoing connections, we do not accumulate the value in order not to increase false-positives. The accumulation is straightforward. For example, a worm registers an autorun entry in the registry in window 0 and drops a backdoor in the system directory in window 1. Suppose the behavior vector's first two fields are $\langle autorun\ key, dll\ drop, \ldots \rangle$. The vector in window 1 will be $\langle 1, 1, \ldots \rangle$ instead of $\langle 0, 1, \ldots \rangle$. This way, even if a worm does only one bad thing in each time window to lower its suspicion level, the suspicion level will finally increase as more malicious activities are exhibited. This mechanism also works for a worm spawning multiple processes with each exhibiting malicious activities in different time windows.

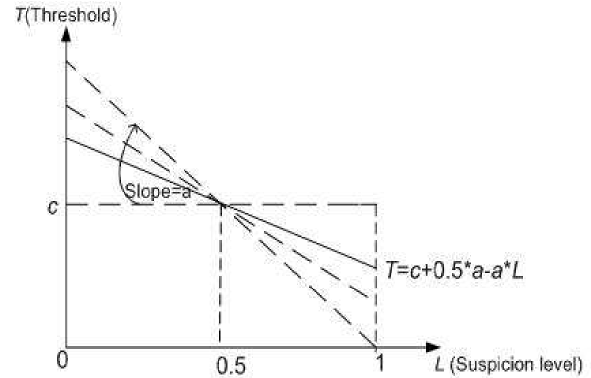

Figure 2: Static threshold vs. per-process threshold

As we do not have such a real-world worm sample available, to evaluate the accumulation scheme we simulated a worm in experiment to illustrate the difference of suspicion level between the original and the accumulated scheme. The results are in Section 6.

We will next present a model and an algorithm used to transform the suspicion level to an appropriate containment threshold.

5. PER-PROCESS CONTAINMENT

The containment scheme seeks to rate-limit the propagation of network worms while allowing for the operation of normal programs on a host. As mentioned earlier, each process's suspicion level is computed for a time window based on the activities observed during the last time window. This suspicion-level information is then mapped to a threshold. The threshold indicates a connection rate value beyond which the outgoing connections will be blocked. The threshold for each process changes as a process's runtime behavior differs during each time window.

5.1 The Mapping Function

The key element in our approach is how to map each suspicion level to a threshold beyond which the process is rate-limited. This mapping function can be of any form but should have a common property; the set of thresholds for processes with higher suspicion levels should be lower and for those with lower suspicion levels should be higher. The rationale behind this is that when the suspicion level for a process is high (low), we would like to block its outgoing traffic as much (little) as possible. The mapping function should therefore be monotonically decreasing within [0,1]. Any type of functions satisfying this property can be used for our purpose. For computation and comparison convenience, linear functions are adopted in our model. Actually, linear functions are found to work well in Section 6. Specifically, we choose $t = h(l) = c + 0.5 * a - a * l, a, c \geq 0, l \in [0, 1]$ as the class of mapping functions (Figure 2), where t denotes the threshold, l is the suspicion level, and a and c are two design parameters. The smaller the a, the less suspicion-level information is used. When the slope $a \to 0$, it is equivalent to using a static threshold to all processes, ignoring the suspicion level. Parameter c reflects the tolerance to the false-positive rate. Given a, the larger the c, the higher thresholds assigned to all processes. Given the form of the mapping function, it is crucial to choose appropriate values of a and c. The criteria for the efficacy of containment are false-positive and false-negative rates. To calculate the false alarm rates, we develop a probabilistic model by assuming certain properties of normal and malicious processes in order to obtain false-positive and false-negative profiles in terms of a and c. Based on the false alarm

Table 2: Connection-rate statistics

	Average Conn Rate	Standard Deviation
Normal	1.75	1.40
Malicious	6.08	5.00

profiles, an optimization algorithm is designed to find the appropriate parameters for the mapping function. Our model and algorithm are presented next.

5.2 Modeling False Alarms

5.2.1 Assumptions

The following assumptions are used in this model.

1. A process is either normal or malicious.
2. The suspicion level across all processes could be treated as a random variable denoted by L_0 (L_1) for normal (malicious) processes, where $L_0, L_1 \in [0, 1]$. $L_0(L_1)$ has cdf $F_0(F_1)$ and pdf $f_0(f_1)$.
3. A mapping function h is from $L(L = L_0$ or $L_1)$ to T (threshold). The threshold for normal (malicious) processes is denoted by T_0 (T_1).
4. First-contact outgoing connection (connection to an address the sender has not recently contacted) rates denoted by R_0 for normal and R_1 for malicious processes follow certain distributions: $F_{conn0}(f_{conn0})$ for normal and $F_{conn1}(f_{conn1})$ for malicious processes. Those connections are of interest because malicious programs tend to reach as many hosts as possible while normal programs have the "locality" property in outgoing traffic.

5.2.2 Data Analysis

We estimated the distributions of first-contact outgoing connections based on real-world traces. For normal programs, we used attack-free network traffic by tcpdump that lasts 17187 seconds, including 585,000 frames. We have selected all new connections initiated, and filtered out other traffic. The connection rate is defined as the number of connections per second. For malicious programs, due to relatively limited access to the real-world network worms, we collect network activities from 10 types of worms and some of their variants. We set up 3 virtual machines connected via a virtual network as our test-bed to collect the network activity data. The connection rate CDFs are shown in Figure 3. We find that 60% of the normal programs' connection rates are around 1/s and 100% of their rates are below 7/s. On the other hand, 50% of malicious programs' connection rates concentrate in the range from 5/s to 8/s. The average and standard deviation across all normal and malicious processes are given in Table 2. Clearly, worm's connection rate is, in general, higher than that of normal programs reflecting the fast propagation of network worms.

5.2.3 False-Alarm Equations

A false-positive occurs when the connection rate of a normal process exceeds its threshold. A false-negative occurs when the connection rate of a malicious process is below its threshold. If the threshold is static for all processes in a host, it can not effectively contain worms and accommodate normal applications at the same time. Our proposed rate-limiting is to assign each process with a customized threshold, which is much finer-grained. To compare it against static threshold approach in terms of false alarms, we derive the false-alarm equations for both approaches. Those equations are

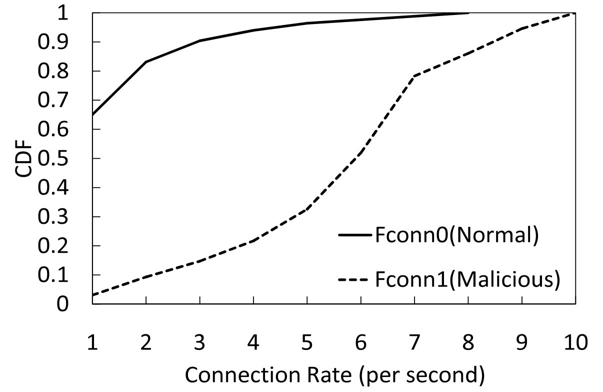

Figure 3: Connection-rate CDFs

represented by the parameters defined in Section 5.2.1. In the static threshold's case, let c denote the constant threshold, then
False-positive: $\Pr(R_0 \geq c) = 1 - F_{conn0}(\lceil c - 1 \rceil)$
False-negative: $\Pr(R_1 < c) = F_{conn1}(\lceil c - 1 \rceil)$
False-positive and false-negative equations for per-process containment are calculated as follows. The threshold in this case is a random variable, rather than a constant, since L is a random variable and $h(L) = T$.
False-positive:

$$\Pr(R_0 \geq T_0) = \Pr(T_0 \leq R_0) = \sum_{r=0}^{\infty} \Pr(T_0 \leq r) \Pr(R_0 = r) = \sum_{r=0}^{\infty} F_{T0}(r) f_{conn0}(r). \quad (1)$$

Note that F_{T0} is the CDF of T_0 when the process is normal and $h(l) = t$. We also know L_0 has CDF F_0 and pdf f_0 for normal processes. By the Change of Variables Theorem [20],

$$f_{T0}(t) = f_0(h^{-1}(t)) \frac{1}{h'(h^{-1}(t))}, \ t \in [h(1), h(0)]$$

$$F_{T0}(t) = \begin{cases} 0 & t < h(1) \\ \int_{h(1)}^{t} f_0(h^{-1}(t)) \frac{1}{h'(h^{-1}(t))} \mathrm{d}t & t \in [h(1), h(0)] \\ l & t > h(0) \end{cases}$$

Similarly, false-negative:

$$\Pr(R_1 < T_1) = \Pr(T_1 > R_1) = \sum_{r=0}^{\infty} \Pr(T_1 > r) \Pr(R_1 = r) = \sum_{r=0}^{\infty} (1 - F_{T1}(r)) f_{conn1}(r). \quad (2)$$

$$F_{T1}(t) = \begin{cases} 0 & t < h(1) \\ \int_{h(1)}^{t} f_1(h^{-1}(t)) \frac{1}{h'(h^{-1}(t))} \mathrm{d}t & t \in [h(1), h(0)] \\ l & t > h(0) \end{cases}$$

Each pair of a and c determines a mapping function. We plug in the connection-rate and suspicion-level distributions as well as the mapping function into the equations (1) and (2) to calculate the false-positive and false-negative rates. Since a pair of a and c corresponds to a pair of false-positive and false-negative, by varying the values of a and c, we can plot a set of false-alarm profiles. As shown in Figure 4, given a, as the value of c decreases from 14 to 1, the curve descends from the upper-left to lower-right direction meaning that the more restrictive the threshold (the smaller the c), the higher the false-positive and the lower the false-negative, showing a tradeoff between the two. Fixing the same set of c from 1 to 14, we vary a's value and generate a curve for each a. When

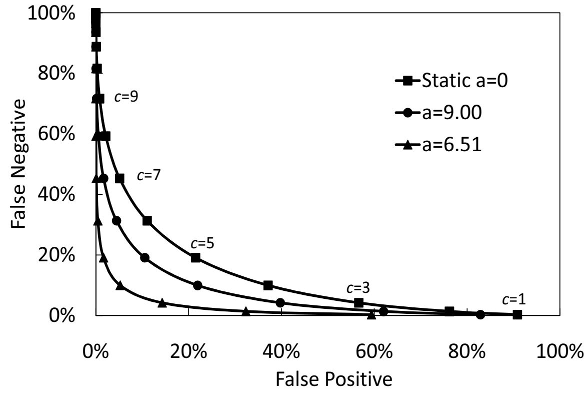

Figure 4: Static threshold vs. per-process false alarm profiles

$a > 0$ (we only draw $a = 6.51$ and $a = 9$ for illustration), meaning that we make use of the suspicion-level information and assign each process a customized threshold, the curves are always below the static threshold approach (i.e., $a = 0$). In other words, given a false-positive rate, the $a > 0$ curves can always achieve lower false-negative rates than $a = 0$ curve does, indicating that using per-process suspicion-level information results in an improved false alarm curve.

5.3 Mapping Function Optimization

To find the most appropriate mapping function for a specific host system, we develop an optimization algorithm. The required false-positive rate is the input to the optimization algorithm that determines a and c to obtain the lowest false-negative rate. We impose a constraint on the false-positive rate because users are affected most by this rate. But this configuration is tunable such that false-negative rate could also be constrained. Since it is difficult to derive explicit equations for a and c, we devise a heuristic algorithm based on the observation of the numerically-obtained false alarm curves. One property of the curve is that given a, the larger the c, the lower the false-positive rate. Another property is that given c, there is an optimal a that could achieve the lowest false-positive. Our algorithm consists of adjustment (steps 1–3) and refinement (steps 4–6). The first phase searches for the curve to the lowest-left direction, while the second phase helps to jump out of the local optimum facing the first phase, if any.

1. (Adjust): given the initial a and c, increase or decrease c to achieve the target false-positive rate.
2. (Adjust): fix c at the value obtained from step 1, increase or decrease a to reach the lowest false-positive rate.
3. (Adjust): repeat steps 1 and 2 until a cannot be changed any further.
4. (Refine): increase or decrease a if a lower false-negative rate can be achieved.
5. (Refine): adjust c to the target false-positive rate.
6. (Refine): repeat steps 4 and 5 until the lowest false-negative rate is reached.

The pseudocode for this is given in Appendix.

6. IMPLEMENTATION AND EVALUATION

We have implemented four behavior monitors: Registry Activity Monitor (RAM), File Activity Monitor (FAM), Network Activity Monitor (NAM), and Process Correlation Monitor (PCM). These monitors capture each process's behavior in real time. The traces collected by these monitors are fed to our trace-driven simulation of the proposed framework. The traces were collected from 20 real-world worms plus some of their variants that are representative and reflect the evolution of contemporary worms and 49 normal programs. We used a C++ implementation of SVM learning algorithm, called LibSVM [21], in the behavior analysis component and derived suspicion-level distributions for normal and malicious processes. We also tested the containment scheme's false-positive and false-negative rates in evaluation.

6.1 System Monitors

The architecture of RAM resembles that of SysInternal's Regmon [22]. RAM was implemented on Windows NT/XP, including a user-level logging application and a kernel device driver which implemented the system-call hooking technique [23]. RAM intercepts registry-related system calls and stores passed parameters and other status information in a kernel buffer which is then periodically copied to the user-level application. RAM logs complete information about every registry activity for all processes running on a host, including timestamp, process name, process ID, request type (create key, set key value, etc.) and path of the registry key. The implementation of FAM is similar to that of RAM. It records system-wide file-system activities in real time. NAM is implemented based on Winpcap library, and continually monitors all incoming and outgoing packets of the host. With Winpcap's support, NAM provides information on active connections (e.g., source address, port, destination address port, process ID, etc.) and dynamically correlates each captured packet with the process that initiates this connection. The data collected by NAM consists of timestamp, process name and ID, connection type (TCP or UDP) and detailed packet header. PCM uses a set of Windows APIs, known as Process Structure Routine, to track process creation, termination and inter-process relationships. These four monitors together characterize the detailed behavior of all the running processes, which will be formalized into behavior feature vectors to determine the per-process suspicion level by the machine learning algorithm.

6.2 Trace Collection

To collect worm traces in real time, we set up 3 virtual machines running Windows XP systems connected via a virtual network as our test-bed. We also set up a DNS server at the host machine to collect DNS statistics and configured it as the default gateway for the virtual machines. By studying recent worm behaviors, we selected 20 real-world worms and their variants. The samples include notable worms such as Blaster, MyDoom, Storm, etc. We ran the worm samples on our test-bed, gathering their process correlation, file system, registry and network activities. The length of trace for each worm is approximately 20 minutes. The normal traces were collected from malware-free PCs in regular use. We selected applications with network access, such as P2P, web browser, file download, etc. The traces captured the activities of 49 normal processes which cover most commonly-used network applications, including eMule, IE, firefox, sshclient, utorrent, etc., and each lasted 20 minutes as well. We did not capture longer traces because most applications show relatively stable behavior. We used part of both normal and worm traces to train the SVM to build profiles and the rest as our testset. We intentionally selected variants of some worm samples into the testset and the original worms in the training set. The accuracy of the learning algorithm with regard to suspicion-level generation is demonstrated in Section 6.4.

6.3 Trace Formalization

Table 3: Suspicion levels with and without feature accumulation

Time Window	Behavior	Original SusLevel	Accumulated SusLevel
win0	Write exe to sys dir	0.59	0.59
win1	Write a dll	0.36	0.68
win2	Create an autorun key	0.20	0.72
win3	Create an Inprocserver key	0.22	0.80
win4	Initiate 20 connections	0.16	0.89
win5	None	0.027	0.89
win6	Initiate 20 connections	0.16	0.89
win7	None	0.027	0.89

We extracted useful features from the file system, registry and network activity logs, and formalized them to feature vectors in a uniform format that can be analyzed by the learning algorithm. A sample format is given as:

0:112 1:0 2:112 3:0 4:112 5:0 6:1 7:0 8:0 9:1 10:0 11:1 12:1 13:0 14:0 ;Bozori worm [00:17:51, 00:18:56]

A feature vector has 15 dimensions, each of which corresponds to an atomic behavior feature represented with a tuple <feature index:value>. For example, the first tuple 0:112 indicates that the process has initiated 112 first-contact connections. A detailed description of the behavior feature vector is given in Section 4.1. As described earlier, we kept track of the process relationships. The behavior features are aggregated across correlated processes. For example, process A registers an autorun entry in the registry and creates process B. Process B then drops a backdoor in the system directory within the same time window. Suppose the behavior vector's first two fields are $\langle autorun\ key, dll\ drop, \ldots\rangle$. Then, the vector in this window for both will be $\langle 1, 1, \ldots\rangle$ instead of $\langle 1, 0, \ldots\rangle$ and $\langle 0, 1, \ldots\rangle$.

In addition, we selectively accumulated some feature fields in consideration of the worms that may spread malicious activities to different time windows. Because we did not find such a worm in the wild, we simulated one to show the difference in suspicion levels. Without feature accumulation, the suspicion level ranges from 0.027 to 0.59, and 7 out of 8 are below 0.5 (Table 3). With feature accumulation over 5 time windows, the suspicion level becomes 0.89. Note that the simulated worm is slowly propagated compared to fast-scanning worms that can generate hundreds of connections in a time window. Even so, due to its accumulated file system and Registry activities, its suspicion level is high enough to trigger a strict rate-limiting when it accesses the network.

For this feature accumulation, we need to determine how long to accumulate behavior for each process. If this time window of accumulation is static, an attacker may learn and evade it. If we accumulate over infinitely many windows, a total number of false-positives may become very high. So, we dynamically change the accumulation time window. For example, we randomly select a value between 1 and 50 min each time. By introducing this uncertainty, it makes evasion of behavior monitoring harder.

6.4 Suspicion-Level Analysis

Recall that the suspicion level generated by the learning algorithm is denoted by L_0 for normal and L_1 for malicious processes where $L_0, L_1 \in [0, 1]$. L_0 (L_1) has CDF F_0 (F_1). We estimated the suspicion-level CDFs for normal and malicious processes by applying the pre-trained SVM to the behavior vectors generated from the normal and malicious traces. The suspicion-level CDFs are plotted in Figure 5. Clearly, normal applications tend to have a lower degree of suspicion, while worms have much higher suspicion levels. This demonstrates the SVM's learning ability in determining the suspicion level of a process, and also indicates that the thus-generated suspicion level is indeed informative.

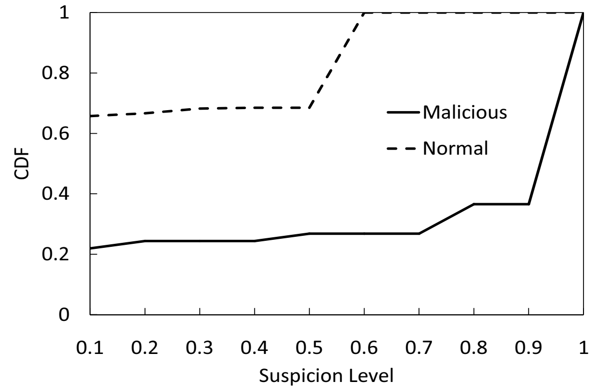

Figure 5: Suspicion-level CDFs for malicious/normal processes

6.5 Overhead

One may want to know the overheads incurred by the runtime behavior capture and the periodical suspicion-level generation. We used a common Windows benchmark PassMark Software, PerformanceTest [24], to measure the overheads of the runtime system monitors on a host machine with Intel(R) Pentium IV 1.5GHz CPU, 512MB memory, 19.5G disk, and Windows XP operating system. We ran the corresponding benchmark program for CPU, memory and disk, respectively, 5 rounds each. The average overhead for CPU is 10.5%, memory 14.5% and disk 4.7%. Considering the fact that memory of this machine is much smaller than that of a today's PC/laptop, all of these numbers are within an acceptable range. As to the suspicion-level generation, since the classifier is pre-trained (i.e., the support vectors are pre-loaded), the training time will not incur any runtime overhead to the host. Calculation of the suspicion level is fast. For example, the suspicion levels of 10 processes are generated within half a second.

6.6 Trace-Driven Evaluation

We simulated the running of different worms and normal processes based on the real traces collected. To demonstrate the efficacy of our scheme, Williamson's rate-limiting [8] was implemented as a baseline in our evaluation. We specifically compared the performance between Williamson's and our per-process schemes when a host was infected by the latest Storm worm. We also applied Williamson's, static-threshold (i.e., processes have the same threshold) and our customized per-process three approaches to all other testset data we collected including worms, their variants and normal programs, to show the performance differences.

6.6.1 Case Study: Storm

Storm worm (or W32.Peacomm, Nuwar, Zhelatin) spreads via email spam and is known to be the first malware to seed a botnet in a P2P manner without any centralized control. It first came out in Jan 2007 and has been active until now. By late 2007, it was estimated to run on around 250,000 to 1 million compromised systems and considered to be a major risk to increase bank fraud, identity theft, and other cyber crimes. Although Storm requires the user to follow a URL in the spam mail to download the executable, the bot dropped into a host system is commanded and controlled via a P2P network. Storm evolves very quickly. The sample we obtained was from the most recent Storm outbreak on Valentine's Day 2008. The trace shows that Storm first connects to the P2P network by contacting peers in a hard-coded peer list containing more than 100

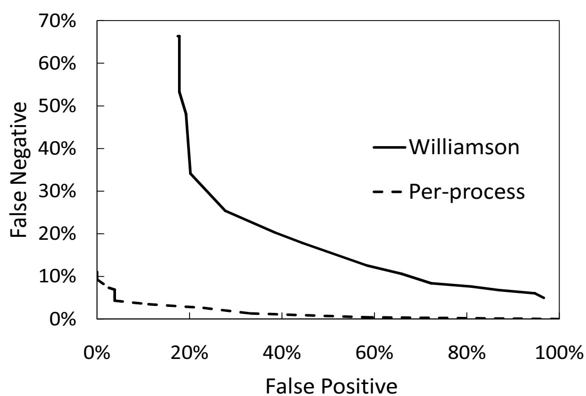

Figure 6: False alarm profiles for Storm worm under Williamson and our approaches

IPs. After joining the network, the bot sends out search requests to find a specific secondary injection for spamming. We observed that it started to behave as a SMTP server and to send spam email in 5 minutes upon execution.

In our experiment, we applied both Williamson's rate-limiting and our per-process schemes. Williamson's approach applied to the entire host. A working set of specified size ($n = 4$ in our case, as commonly used by others) is maintained to keep track of all IPs the host has contacted. When a new connection is initiated, the destination IP is compared with those in the working set. If it is in that set, the connection can pass through. Otherwise, it is placed in a delay queue and will be sent out later. At periodic intervals (every second as Williamson proposed), one connection is dequeued and a new destination address is added to the working set. There is a pre-determined threshold for the delay queue. Whenever this value is exceeded, all new connections are dropped.

We mixed the normal trace including P2P, web browser applications with storm trace in the simulation of Williamson's per-host scheme. We varied the delay queue threshold in each round to get a pair of false-positive and false-negative rates. In our per-process scheme, we used the mapping function $h(l) = c{+}0.5{*}a{-}a{*}l$. The optimal mapping is generated by the algorithm described in Section 5. We obtained several pairs of false-positives and false-negatives via different mapping functions and then drew the false alarm profiles. Figure 6 compares false alarms of the two approaches, showing that our scheme outperforms Williamson's. In Williamson's scheme, when the delay queue threshold is set to a small value, Storm can saturate the delay queue in tens of seconds, thus causing normal traffic to be dropped. When there are some network-intensive normal applications, the false-positive is considerable. Specifically in the experiment, the false-negative rate is controlled within 10% at the expense of more than 70% false-positive rate. On the other hand, given a generous delay queue threshold to accommodate normal traffic, a majority of Storm's connections can pass through too. The problem lies in treating all processes indiscriminately. In per-process scheme, different thresholds are assigned to different processes according to their suspicion levels. In the normal case, the average suspicion level for all normal applications is around 0.3 despite that some network-intensive applications are included. On the other hand, Storm has acted maliciously at the Registry and file system resulting in a suspicion level of 0.95 and hence a low rate-limiting threshold. By imposing customized rate-limiting to Storm and normal traffic, our scheme achieves lower false-positive and false-negative rates.

6.6.2 Evaluation of Three Schemes

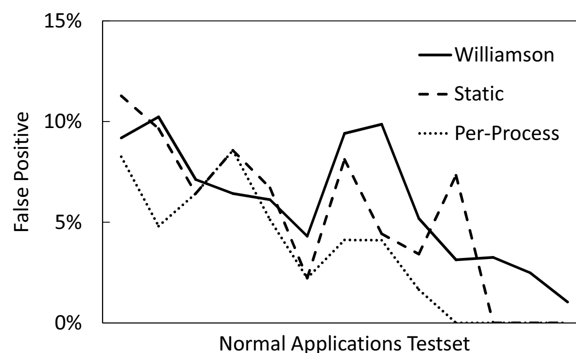

Figure 7: False-positives on normal applications

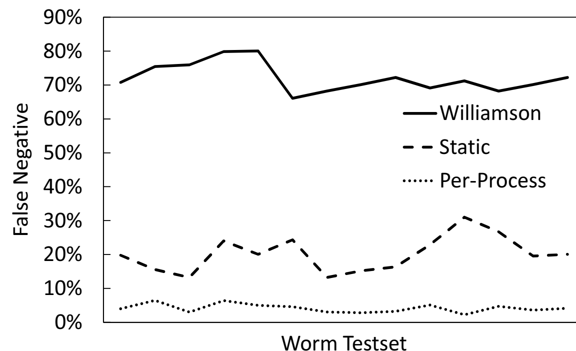

Figure 8: False-negatives on worms

Note that Williamson's use of a per-host static threshold is slightly different from the static-threshold approach mentioned before. The latter assigns the same threshold to all processes, while the former does not discriminate at the process level. To compare these two and our schemes, either false-positive or false-negative rate has to be fixed as there is always a tradeoff between the two. We set the false-positive rate to be 5% and input the parameters to the optimization algorithm. The resulting optimal mapping function, which can generate the lowest false-negative rate is $h(l) = 7.32 + 0.5 * 6.51 - 6.51 * l(a = 6.51, c = 7.32)$. Based on the numerically-obtained false alarm profiles introduced in Figure 4, the static threshold rate is set to 8 per second to meet the false-positive rate requirement. Since a suspicion level is generated for each process every t minutes (t is set to 1 in our experiment), the threshold value calculated by the mapping function is also updated dynamically in the order of minutes. While applying Williamson's scheme, we chose the well-adopted parameters (working set length=4 and release rate of delay queue= 1 connection/s). We varied the delay queue threshold imposed to mingled normal and worm traces to find the one that can reach a 5% false-positive rate. The appropriate value is found to be 200.

Recall that the false-positive rate is defined as the fraction of normal connections blocked. Figure 7 plots real false-positive rates for each normal program in the testset under Williamson's, the static threshold, and the customized per-process schemes. They have some fluctuations process-wise but the average values are close to 5% (see Table 4). Figure 8 shows the false-negative curve for each scheme. The false-negative rate is calculated as the percentage of evaded connections of worms. Given false-positive rates shown in Table 4, the average false-negative rate across all worms under Williamson's scheme is the highest, 72.12%, and the per-process scheme 4.15%, the best. The static scheme produces 20.14%, in the middle. Williamson's is even worse than the static threshold scheme for the following reason. During a worm outbreak, the per-

Table 4: Average FPs and FNs under three schemes

	Avg FP	Avg FN
Williamson	5.56%	72.12%
Static	4.87%	20.14%
Per-Process	3.24%	4.15%

host delay queue is mostly occupied by the worm. When the threshold is set to a larger value to accommodate normal applications, the worm benefits more, leading to a high false-negative rate. As for the static threshold scheme, the threshold is assigned to each process, which is relatively small (in our case, 8 versus 200) and thus more restrictive, compared to the per-host threshold.

Although progress is made from the per-host to the static threshold scheme, our customized per-process threshold can perform even better than the static threshold scheme by using suspicion-level information. The suspicion levels for worms are relatively high and their thresholds are accordingly low so that most of malicious connections may be blocked, whereas normal applications are assigned low suspicion levels but high rate-limiting thresholds to let most traffic pass through. Compared to the customized threshold, the static threshold scheme must compromise one false alarm rate for another. Thus, the customized per-process scheme performs best among the three.

6.6.3 Optimality of the Mapping Function

We now want to show that the performance improvement from use of a static threshold to a customized per-process threshold is not a coincidence and that the mapping function used is optimal in the sense that it can achieve the lowest false-negative rate given a false positive rate. We selected 12 pairs of a and c, i.e., 12 mapping functions, and measured the false-positives and false-negatives on the same data set. Figure 9 plots the false-alarm profiles for different a and c values based on the real-world traces. As we expected, the curves are quite similar to the numerically-computed false-alarm profiles (Figure 4). The curve in the lower-left direction is the one with the optimal a because given a false-positive rate, the false-negative rate on this curve is always lower than other curves, and it is the same case when false-negative is fixed. In this figure, $a = 6.51$ is obviously better than smaller or larger values of a. This is the value our optimization algorithm yielded. In particular, on this curve, $c = 7.32$ is the one close to our required false-positive rate 5%. This c value is also identical to that generated from the optimization. Moreover, when $a = 0$ which represents the static scheme, the false-negatives on this curve are generally higher than those on other curves. All of these observations confirm that our empirical results obtained from real-world traces are consistent with the numerical results obtained from false-alarm modeling, indicating that the mapping function selected by the optimization algorithm is indeed optimal and the per-process scheme performs significantly better than the static threshold scheme in terms of false-alarm rates.

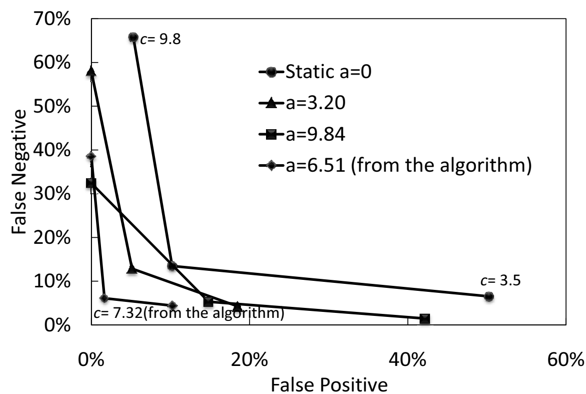

Figure 9: Optimality of the mapping function

7. LIMITATIONS AND FIXES

In this section, we would like to discuss two fundamental limitations of not only our scheme but all host- and behavior-based worm defense and response mechanisms. The first limitation is the circumvention of a pre-defined list of behaviors. Since our behavior list that can best discriminate normal and malicious programs is based on the study of existing network worms, our scheme works effectively for malicious processes having typical "worm" behavior. Even if some worms change their behaviors a little bit, such as installing in a different directory, our scheme can still work since we account for a *set* of behavior features, not just one feature. As worms evolve, we can simply extend or modify our behavior list of monitoring. However, if all of the behaviors of a worm are the same as those of normal programs or completely different from existing worms, we can hardly capture it. However, such a scenario will be rare as our behavior list reflects the fundamentals of network worm behavior.

The second limitation is the vulnerability of a host-based mechanism to worms' adaptivity. There is always a tradeoff between deploying worm defense at the network and at end-systems. A network-based scheme is not easy to be disabled by a worm but only gets coarser-grained information, i.e., network activities on a host basis and can be fooled by address spoofing, resulting in less accurate and efficient response. A host-based solution, on the other hand, can obtain finer-grained information and achieve finer-grained and accurate response as we have demonstrated. The way a worm gets around our scheme is to sit below the monitoring level and modify or subvert the information our monitor receives by using the rootkit technique for example. One countermeasure is to search for the discrepancy between the information returned by the Windows API or system calls and that seen in the raw scan of the file system or Registry hive [25]. With the help of secure hardware [26] or secure VMM [27], it is also possible to prevent or detect the rootkit from altering the OS.

While there is a possibility that a worm could attempt to evade our mechanism, in the evaluation section we have demonstrated how our system successfully contains state-of-the-art worms that we were able to collet while minimizing the impact on legitimate traffic. Therefore, our approach at least raises the bar significantly for contemporary network worms.

8. CONCLUSION

We have proposed a novel automatic worm defense framework that combines per-process behavior analysis and fine-grained containment. It automatically monitors each process's runtime behavior and generates its level of suspicion by a machine learning algorithm. A mapping algorithm is developed to transform the suspicion level to the appropriate rate-limiting threshold on a per-process basis. Our experimental evaluation based on real-world worm samples and normal process traces demonstrates the efficacy of per-process rate-limiting, which produces much fewer false-positives and false-negatives in containing network worms than previously-known approaches.

Acknowledgements

The work reported in this paper was supported in part by the US National Science Foundation under Grant CNS-0523932.

9. REFERENCES

[1] Symantec. Symantec internet security threat report highlights rise in threats to confidential information. www.symantec.com/press/2005/n050321.html, 2005.

[2] M.Christodorescu, S.Jha, S.A.Seshia, D.Song, and R.Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, 05.

[3] E.Kirda, C.Kruegel, G.Banks, G.Vigna, and R.Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. *IEEE Symposium on Security and Privacy*, 120, 1996.

[5] A. Somayaji and S.Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Security Symposium*, 2000.

[6] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[7] H.Wang, S.Jha, and V.Ganapathy. Netspy: Automatic generation of spyware signatures for nids. In *Proceedings of Annual Computer Security Applications Conference*, 2006.

[8] M.Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of The 18th ACSAC*, 2002.

[9] S.Chen and Y.Tang. Slowing down internet worms. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, March 2004.

[10] S. E. Schechter, J. Jung, and A. W. Berger. Fast detection of scanning worm infections. In *Proceedings of RAID*, 04.

[11] V.Sekar, Y.Xie, M.K.Reiter, and H.Zhang. A multi-resolution approach forworm detection and containment. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[12] W.Lee, S.Stolfo, and K.Mok. A data mining framework for building intrusion detection models. In *Proceedings of IEEE Symposium on Security and Privacy*, 1999.

[13] S.Singh, C.Estan, G.Varghese, and S.Savage. The earlybird system for real-time detection of unknown worms. Technical report, University of California, San Diego, 2003.

[14] T.Lee and J.J.Mody. Behavioral classification, 2006.

[15] C. Wong, S. Bielski, A. Studer, and C. Wang. Empirical analysis of rate limiting mechanisms. In *Proceeding of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[16] D.Whyte, E.Kranakis, and P.V.Oorschot. Dns-based detection of scanning worms in an enterprise network. In *Proceedings of NDSS*, 2005.

[17] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[18] T. Joachims. Making large-scale support vector machine learning practical. In B. Scholkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[19] H. Lin, C. Lin, and R. Weng. A note on platt's probabilistic outputs for support vector machines, 2003.

[20] Change of variables theorem. http://mathworld.wolfram.com/ChangeofVariablesTheorem.html.

[21] C.-C. Chang and C.-J. Lin. Libsvm – a library for support vector machines. www.csie.ntu.edu.tw/ cjlin/libsvm/.

[22] Regmon. www.microsoft.com/technet/sysinternals /utilities/regmon.mspx.

[23] Api hooking revealed. http://www.codeproject.com/ system/hooksys.asp.

[24] Passmark. http://www.passmark.com/.

[25] Rootkitrevealer. http://www.microsoft.com/technet/sysinternals/Utilities /RootkitRevealer.mspx.

[26] Lagrande technology architectural overview, 2003.

[27] T.Garfinkel, B.Pfaff, J.Chow, M.Rosenblum, and D.B.Terra. A virtual machine-based platform for trusted computing. In *Proceedings of the Symposium on Operating Systems Principles*, 2003.

APPENDIX

The mapping function optimization algorithm

ADJUST_C($\&a, \&c, F_{conn0}, targetFP$)
1 **if** FP(a, c, F_{conn0}) $> targetFP$
2 **then** $step \leftarrow EPSILON$
3 **else** $step \leftarrow -EPSILON$
4 **while** FP(a, c, F_{conn0}) $> targetFP$
5 **do** $c \leftarrow c + step$ *or* $c \leftarrow c - step$
6 ADJUST_A($a, c, F_{conn0}, targetFP$)

ADJUST_A($\&a, \&c, F_{conn0}, targetFP$)
1 $currenta \leftarrow a$
2 $currentFP \leftarrow$ FP(a, c, F_{conn0})
3 **if** FP($a + step, c, F_{conn0}$) $< currentFP$
4 **then** $step \leftarrow EPSILON$
5 **else** $step \leftarrow -EPSILON$
6 **while** FP($a + step, c, F_{conn0}$) $< currentFP$
7 **do** $a \leftarrow a + step$
8 $currentFP \leftarrow temp$
9 **if** $a - currenta < EPSILON$
10 **then** $return$
11 ADJUST_C($a, c, F_{conn0}, targetFP$)

REFINE_A($\&a, \&c, F_{conn0}, F_{conn1}, targetFP$)
1 **repeat**
2 $currentFN \leftarrow$ FN(a, c, F_{conn1})
3 **if** FN($a + EPSILON, c, F_{conn1}$) $< currentFN$
4 **then** $a \leftarrow a + EPSILON$
5 **if** FN($a - EPSILON, c, F_{conn1}$) $< currentFN$
6 **then** $a \leftarrow a - EPSILON$
7 **if** FN($a + EPSILON, c, F_{conn1}$) $== currentFN$
8 **then** $return$
9 REFINE_C($a, c, F_{conn0}, targetFP$)
10 **until** FN(a, c, F_{conn1}) $> currentFN$

REFINE_C($\&a, \&c, F_{conn0}, targetFP$)
1 **if** FP(a, c, F_{conn0}) $> targetFP$
2 **then** $step \leftarrow EPSILON$
3 **else** $step \leftarrow -EPSILON$
4 **while** FP(a, c, F_{conn0}) $> targetFP$
5 **do** $c \leftarrow c + step$ *or* $c \leftarrow c - step$

The EPSILON is set to be 10^{-2} and the call sequence is:
adjust_c(a,c,F_{conn0},targetFP)
refine_a(a,c,F_{conn0}, F_{conn1},targetFP)