Inoculation Against Malware Infection Using Kernel-level Software Sensors

Raymond Canzanese and Moshe Kam
Dept. of Electrical and Computer Engineering
Drexel University
Philadelphia, PA, U.S.A.
{rcanzanese,kam}@minerva.ece.drexel.edu

Spiros Mancoridis
Dept. of Computer Science
Drexel University
Philadelphia, PA, U.S.A.
spiros@drexel.edu

ABSTRACT

We present a technique for dynamic malware detection that relies on a set of sensors that monitor the interaction of applications with the underlying operating system. By monitoring the requests that each process makes to kernel-level operating system functions, we build a statistical model that describes both clean and infected systems in terms of the distribution of data collected from each sensor. The model parameters are learned from labeled training data gathered from machines infected with canonical samples of malware. We present a technique for detecting malware using the Neyman-Pearson test from classical detection theory. This technique classifies a system as either clean or infected at runtime as measurements are collected from the sensors. We provide experimental results that illustrate the effectiveness of this technique for a selection of malware samples. Additionally, we provide a performance analysis of our sensing and detection techniques in terms of the overhead they introduce to the system. Finally, we show this method to be effective in detecting previously unknown malware when trained to detect similar malware under similar load conditions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*

General Terms

Reliability, Security

Keywords

System monitoring, malware detection, fault tolerance

1. INTRODUCTION

Malware is software that is designed to infiltrate, damage, or otherwise compromise the computer systems and networks on which it executes. Commercially available antivirus software traditionally adopts a signature-based approach to malware detection, wherein code residing in persistent or volatile memory is labeled as malicious if a subset of the code matches a signature in the detection database. Such antivirus software has seen a widespread adoption due to their low false-positive rate and ease of use [31].

However, the signature-based approach to malware detection is a reactive approach that requires malware first to be discovered and analyzed so that a signature can be added to a detection database. As a result, malware authors modify existing malware to evade detection. This is accomplished through obfuscation, wherein the software is modified by reordering, encrypting, or packing the code, inserting meaningless code, or otherwise changing the structure of the program without altering its function [21]. Such obfuscation can generally change the code enough to evade detection while still maintaining functionality. As an example, the ZeuS botnet was first discovered in 2007 and signatures to detect its many variants are included with most commercially available antivirus software. Nevertheless, by modifying the malware to avoid detection, ZeuS was used in a successful cyber-attack against U.S. government agencies in December 2010 [16]. Furthermore, a family of malware known as metamorphic or polymorphic viruses are also able to elude detection by automatically modifying themselves as they propagate [26].

Because antivirus software systems are reactive, they leave hosts vulnerable to both new malware and to modified versions of previously discovered malware. In environments traditionally targeted by zero-day malware, a more robust approach to detection is desired; specifically, one that is able to detect previously undiscovered malware as it appears on a host system. This paper describes an approach to solving the malware detection problem using a custom sensor suite and a Neyman-Pearson likelihood ratio test.

We developed a sensor suite to monitor the interaction between the applications running on a host and the underlying operating system. The sensors monitor access to operating system functions in Microsoft Windows XP [3] and are implemented as a kernel-level device driver that measures the number of calls made per second to each of the functions. We present a statistical model that describes both clean and infected systems in terms of the distribution of the data collected by the sensors.

The detection technique is comprised of distinct training and detection phases. During the training phase, the sensor suite is used to monitor calls to system functions made by all

ICAC-11, June 14-18, 2011, Karlsruhe, Germany.

of the processes running on the system. The parameters of the model are estimated from labeled training data and the resulting models are used in the detection phase, wherein the detection algorithm classifies the system as either clean or infected.

We present a case study to demonstrate the advantages of using our sensing and detection techniques for malware detection. This study includes seven malware samples that are used to infect a PC running Windows XP and acting as an HTTP server. We present the performance of the detector in terms of its empirical Receiver Operating Characteristic (ROC) and show that the detector is successful in both detecting malware it was trained to detect and similar malware samples that were not used in training. We show that training the models with canonical examples of malware is a successful inoculation technique that enables the system to detect other samples of similar malware without explicitly being trained to detect those samples. Furthermore, we show the detector to be successful in detecting malware under a different type of load intended to mimic desktop-computing usage.

Finally, we present a study of the performance of both the sensors and detector in terms of the overhead they introduce to the system. We present the performance of our system under various benchmarks and compare the results to a common commercial virus scanner.

The remainder of this paper is organized as follows: Section 2 describes related work in malware detection, Section 3 describes the sensor suite we developed to detect malware, Section 4 describes the malware detection technique, Section 5 describes a case study involving the Apache web server, Section 6 presents an analysis of the performance overhead of the system, Section 7 presents the results of the case study, Section 8 addresses threats to validity, and Section 9 states our conclusions and our plan for future work.

2. RELATED WORK

The research community has proposed solutions for malware detection that generally fall into one of three categories: signature-based, anomaly-based, or classification-based [9].

2.1 Signature-based malware detectors

Signature-based detectors include most commercial antivirus software, such as products offered by Symantec [7], and common intrusion detection software such as SNORT [24]. Signature-based approaches are limited because they require detection signatures for known threats to be added to a detection database. As a result, these detectors are vulnerable to zero-day attacks and mutations of known malware, including metamorphic and polymorphic viruses. Recent research has shown that signature-based detectors can be effective in detecting polymorphic viruses [36] and obfuscated variants of known malware [22]. However, these approaches are limited to known polymorphisms and obfuscations.

2.2 Anomaly-based malware detectors

Anomaly detectors are used because they are not subject to zero-day attacks or mutations. Anomaly detectors require accurate models of known system behavior and flag variations from those models as potentially malicious. Sekar *et al.* [28] present an anomaly-based detector wherein the system calls required by a program are manually specified and the detector monitors system calls, flagging any calls outside the specification as malicious. Anomaly detectors are also commonly used in intrusion detection systems, and typically focus on only certain types of attacks, such as web based attacks [17, 34]. Another anomaly-based approach to software fault detection proposed by Stehle *et al.* [30] uses software sensors and computational geometry to detect a variety of software faults, including malware infections.

Anomaly-based detectors are limited because they require accurate models of normal system behavior to be constructed. Because deviation from their models is labeled as malicious, these detectors are also prone to a high rate of false alarm and do not perform well if the software is modified without updating the model of normal behavior.

2.3 Classification-based malware detectors

Classification-based detectors differ from anomaly-based detectors in that they use models of both normal and malicious behavior to detect malware. Classification-based detectors typically use labeled training data or expert knowledge to build models of system behavior. In order to detect the presence of malware, these detectors do inference on the models to determine whether a sample is clean or infected, typically using common machine learning algorithms. Early research in the field of malware detection showed that extracting behavior-related information such as system calls and strings from static executables could be used to train simple detectors such as the naive Bayesian detector to detect malware effectively in the absence of obfuscation [27].

More recent research shows that data gathered from performance monitors can be used as features for a malware detection system [29, 20]. These systems use memory, CPU, network, and power sensors to infer the presence of malware using machine learning approaches.

Research has shown that sandboxing executables and monitoring security-relevant system functions and their arguments can provide a valuable feature set for malware detection that can be classified using Support Vector Machines (SVM) [23]. Furthermore, n-gram analysis of system calls and their arguments can also be effective for malware detection [18]. Martignoni *et al.* [19] show how this type of monitoring can also be used to unpack obfuscated code.

System call monitoring is also used in classification approaches to malware detection based on taint analysis [35, 15, 12]. In taint analysis, programs are executed in an environment that monitors all sensitive information introduced to the system. By monitoring how this information is used, the classifier can infer whether a program is exhibiting known malicious behavior.

These proposed classification-techniques are limited in different ways. Techniques that require system emulators or sandbox environments restrict analysis to be performed offline. Many classification-based approaches also only show their detectors to be robust in detecting known malware, although some efforts have shown their models to be robust in detecting unknown malware [23]. Finally, some efforts have focused on detecting malicious applications running at the user-level, for example by white listing operating system services, which can be problematic for malware that masquerades as an operating system service [18].

The malware detection technique presented in this paper is a classification-based approach that relies on dynamic analysis, specifically by monitoring calls to operating system functions. This technique can be performed on a live

system, and does not require a system emulator or sandbox environment. Furthermore, we do not restrict the number of system calls we are monitoring nor do we exclude operating system services. Rather than rely on sequences of calls or n-gram analysis, we analyze the distribution of the data collected from the sensors in terms of calls-per-second to each function, and use each sensor to monitor the entire system rather than specific processes. Finally, we show that our detector is robust in detecting zero-day malware and is able to successfully detect malware with a low probability of false alarm under two different types of load conditions.

3. SENSING TECHNIQUE

To detect malware, we developed a sensor suite for Microsoft Windows XP that monitors the interaction between software applications and the operating system. The suite consists of 282 sensors, each of which monitors calls made to a single operating system function. The sensors log the process from which the system call originated and report the number of calls made per second to each function. The data from each of the sensors are used to build a statistical model to describe a clean and an infected system and to infer whether a system is infected based on the developed models.

The system functions that the sensors monitor can be classified according to the functions they perform [10]. The categories of the functions include: memory, processes, threads, files, registry keys, security and auditing, and ports. By monitoring access to these functions, we are monitoring how applications interact with the operating system. This sensing technique is relevant to malware detection because malware commonly focuses on stealing data, providing discreet remote access, avoiding detection, and propagation. Each of these tasks require the malware to interact with the operating system in order to access the network, file system, registry, and other system resources to complete their malicious tasks.

Thus, these sensors were developed under the assumption that data gathered from a system infected with malicious software will, by virtue of a the malware's access to system functions for malicious purposes, differ from data gathered from a clean system in ways that can be detected via methods from classical detection theory. Because they are implemented at the kernel-level, the sensors will log malicious access to system functions irrespective of the obfuscation techniques used by the malware.

3.1 Sensor architecture

The Microsoft Windows XP operating system has 284 distinct native operating system functions, many of which are exported for use by third-party developers and others that are used only internally by the operating system. The address of each of these system functions is stored in kernel memory in a data structure called the System Service Dispatch Table (SSDT) [11]. To describe how this table is used, we will consider the example of a user-mode application attempting to write data to a file. When an application calls the `write()` function, the appropriate code in the library for writing files will execute in the context of the application's process. When the library is ready to write to the file, it will call the kernel dispatcher, via either a `SYSENTER` or `INT 2E` interrupt command. The kernel dispatcher, KiSystemService, is triggered by this command and copies the arguments from the application to the kernel. It subsequently looks up the address of the appropriate system function by its index, in this case NtWriteFile (index 0x0112). The dispatcher calls NtWriteFile and copies the returned data back to the application's memory.

In order to monitor access to system functions such as NtWriteFile, our sensors use a technique known as system-call hooking [25], a technique often used by malware authors to avoid detection [14]. This technique works by overwriting the addresses of each system function stored in the SSDT with the addresses of our corresponding sensor functions. Each sensor function logs the numeric process identifier for the process from which the system call originated, the location of the file image from which the process was started, and the time of the call. After this information is logged, the sensor function calls the system function originally pointed to by the SSDT and returns the data from this call to the dispatcher. Thus, the monitoring is transparent to the applications being monitored.

The sensor functions are monitored by a separate thread running at the kernel-level. This thread collects the data logged by the individual sensors and computes the number of calls made to each system function per second. This thread is also capable of logging a record of each system call to a file. The output of this thread is a vector of measurement data, where each entry of the vector is the number of calls made to a particular operating system function in that time period. The kernel thread computes and outputs a new feature vector every second while the sensors are running.

The device driver is controlled by a user-mode application that can install and uninstall the driver and start and stop the sensors. The user-mode application can also communicate directly with the driver to retrieve sensor data for additional processing. Figure 1 provides a visual depiction of the sensor suite's architecture. The components that comprise the sensor suite are highlighted in gray. The sensors are installed in the kernel between the KiSystemService dispatcher and the system functions themselves. Each sensor sends its collected data to a kernel thread that processes the data, calculates the number of calls per second, and logs the data. This monitor thread also communicates with the user-mode controller to exchange data and manage the state of the sensors.

4. MALWARE DETECTION TECHNIQUE

The malware detection system adopts a classical statistical approach to detection, the Neyman-Pearson test [32]. Because the input features are the frequencies of calls to each system function, the detection approach is unaffected by attempts to avoid detection by means of obfuscation, encryption, or polymorphism. Furthermore, since the input features profile the behavior of the applications running on a host by monitoring their interaction with the operating system, this approach is also robust to detecting previously unknown samples of malware that are behaviorally similar to previously discovered samples.

The first step to implementing a Neyman-Pearson detector is to develop a statistical model to describe the data collected from the sensors. Figure 2 shows the marginal distribution of a representative sensor. This sensor measures the number of calls made to the function `NtQuerySystemInformation()`, a function used to query performance counters, get information about processes running on the system, and

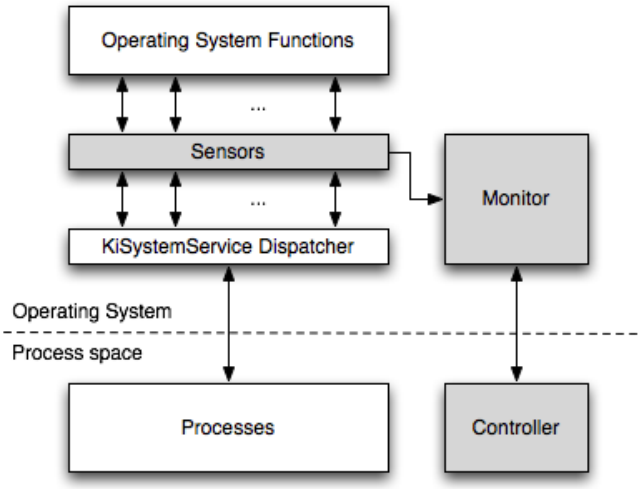

Figure 1: Sensor block diagram showing the components of our sensor system highlighted in gray

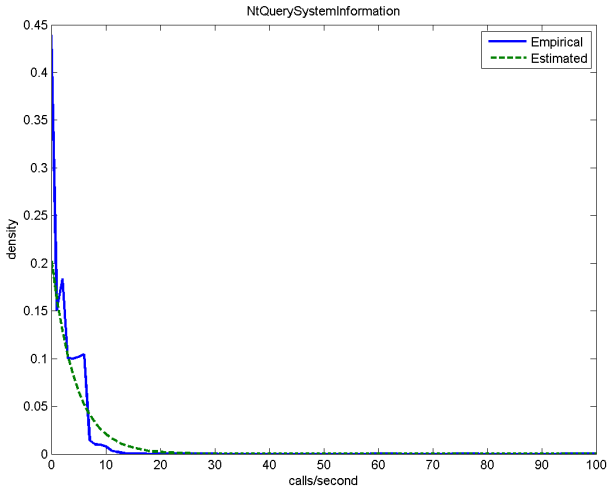

Figure 2: Representative sensor distribution for NtQuerySystemInformation, with a dashed-line indicating a geometric fit to the experimental data

retrieve other system information. As shown in Figure 2, we can approximate the distribution of the senor data as a geometric distribution. To verify that every sensor can be approximated by a geometric distribution, we perform Chi-squared goodness-of-fit tests on data gathered from each of the sensors of the hypothesis that the data are geometrically distributed with parameter p estimated from the data.

Using a geometric distribution for each sensor as the statistical model of both a clean and infected system, we can formulate the detection problem as our sampled data belonging to one of two hypotheses: Either the data are sampled from an infected system or the data are sampled from a clean system. In order to continue, we make the assumption that the data collected from each of the sensors are independent. Although covariance analysis of the sensors indicates this assumption to be false – for example `NtOpenKey`, `NtCloseKey`, and other registry related functions are correlated – we continue with this assumption for a variety of reasons. First, empirical research has shown that classifiers operating under strong independence assumptions, such as the naive Bayes classifier, perform well under many circumstances, including malware detection [13, 33]. Second, assuming independence allows us to develop a computationally simple detector, which is necessary for detecting malware on a live system with minimal overhead. Finally, a detector built under independence assumptions will allow us to gather baseline information of the performance of a simple detector using our newly developed sensors that can be used in future work to compare with other detectors.

To formulate a decision rule to classify data as either clean or infected, we use the following parameterization of the geometric probability mass function (PMF):

$$P(x) = (1-p)^x p, \tag{1}$$

where the support of x is the set of non-negative integers and p is the parameter of the distribution.

We formulate a Neyman-Pearson test for each individual sensor by comparing the likelihood ratio under the two hypotheses to a threshold η_i, particularly:

$$\eta_i \gtrless \frac{P(x|p_{c,i})}{P(x|p_{i,i})} = \frac{(1-p_{c,i})^x p_{c,i}}{(1-p_{i,i})^x p_{i,i}} \tag{2}$$

where $p_{c,i}$ and $p_{i,i}$ are the values of parameter p for a clean and infected system for the i^{th} sensor, respectively. We decide on the threshold η_i by fixing the probability of false alarm and maximizing the probability of detection.

Since we assume the sensors are independent, we can express the likelihood that the data belongs to a particular class as the product of the likelihoods that it belongs in that class for every sensor. By taking the logarithm of this expression, we can express the log likelihood that the data comes from a particular class as the sum of the log likelihood ratio from each sensor. By simplifying this expression, we obtain our decision rule:

$$\eta \gtrless \sum_{i=1}^{282} x\left(\log(1-p_{c,i}) - \log(1-p_{i,i})\right) = \Lambda(x). \tag{3}$$

In order to perform detection, we must first train a model to determine our parameters p of the distribution. For training, we use the maximum likelihood estimate for p. Because our formulation of the geometric distribution allows the sensor values to be zero, the maximum likelihood estimator for p is the reciprocal of the sample mean offset by one:

$$\hat{p} = \left(1 + \frac{1}{n}\sum_{i=1}^{n} x_i\right)^{-1}. \tag{4}$$

For training purposes, all data must be labeled as either clean or infected. For each set of labeled data, we calculate the estimate $\hat{p}$ for each sensor. Thus, the output of the training phase is two vectors that parameterize the model used for detection.

During the detection phase, data from the sensors and the $\hat{p}$ values learned from the training data are input to equation 3 and if the likelihood ratio exceeds the threshold it is labeled 'clean', otherwise it is labeled 'infected'. It is important to

note that the decision rule is a simple weighted sum and therefore can be calculated extremely quickly. Furthermore, the decision rule as described in Equation 3 uses only one sample taken from each sensor. To improve detection, we can train a model that uses N samples from each sensor for detection. If we treat successive samples from each sensor as independent, we can formulate the Neyman-Pearson test as the sum of the likelihood ratio test given in Equation 3 over N samples:

$$\eta \gtrless \sum_{j=1}^{N} \Lambda(x_j). \quad (5)$$

Thus, Equation 5 describes the detector that we will use to infer the presence of malware.

5. EXPERIMENTAL SETUP

This section provides an overview of the case study that was performed to determine the effectiveness of the sensing and detection techniques described in this paper. All tests were performed on a Dell Optiplex GX280, with a Pentium 4 2.8GHz processor and 1GB of RAM, running Microsoft Windows XP Professional, Service Pack 3.

The testbed is configured to operate as a webserver running the Apache 2.2 webserver [1], the PHP 5.3 hypertext processor [5], and the MySQL 5.5.8 community database server [4]. Additionally, the Microsoft .NET framework versions 2, 3, and 4 are installed since they are frequently required for malware to run successfully. The Microsoft Office 2007 suite is also installed, as it is another common vector of attack.

The webserver hosts two separate websites: one using the Drupal 7 [2] content management system (CMS) and the other using the Silverstripe 2.4 [6] CMS. The Drupal installation is populated only with the default content, while the Silverstripe installation contains a copy of the course website for the graduate-level systems sequence ECES511-ECES513 at Drexel University. This site includes course information, links to resources, homework assignments, lecture notes, and video lectures.

Since the goal of this paper is to characterize the performance of our detection technique in a realistic scenario, we use both the Drupal and Silverstripe websites to place a workload on the system. For the Drupal website, a remote host connects to the management interface of the site and requests the the full unit test suite be performed on the server. The server then begins the test suite, reporting results of the tests back to the client as they complete. These tests include functional tests such as the creation, modification, and deletion of content, and system tests, such as the execution of infinite loops.

For the Silverstripe website, we collected Apache web server access logs for the live version of the course website over an 11 week period, from September to December 2010. Over this time period, the data from a typical week consisted of approximately 1000 pageviews from 135 unique visitors.

After installing our sensors on the testbed, we run both the Drupal unit test suite and the web-access replayer from a remote host and log the data from the sensors. The data collected from this series of tests are used to build a model of a clean system. To build a model of an infected system, we repeat the tests on a system infected with various samples of malware.

In total, we gathered two clean data sets and seven infected data sets, one for each malware sample, each 16 hours in length.

5.1 Malware samples

This section briefly describes the seven malware samples used for testing. These samples can be classified as worms, trojans, and botnets.

5.1.1 Worms

Autoit and *YahLover* are both network worms written in the Autoit scripting language. Each of these samples propagate via popular messaging protocols and removable media. YahLover is much more aggressive in infecting removable media, writing copies of itself to every folder on the removable media and disguising the copies as folders.

5.1.2 Trojan Horses

Bybz is a Trojan horse that spreads via removable media and over unprotected network shares. It steals user data and transmits them via a backdoor to a remote host. It also blocks access to security-websites. *Bohu.a* is a similar Trojan, disguised as a Chinese-language high definition video player that also blocks access to security-related websites. It also blocks network traffic from common antivirus software. *Carberp* steals user data such as browsing histories, and transmits them via a backdoor to a remote host.

SpyEye is an advanced Trojan horse of Russian origin, similar to the ZeuS malware. It monitors keystrokes, email, and http access in an attempt to steal passwords and other data, and transmits the data to a remote host. SpyEye is part of a toolkit available for purchase on illicit channels that allows malicious users to craft instantiations of the malware for their specific uses. SpyEye uses rootkit-like functionality to disguise itself and prevent detection.

5.1.3 Botnets

Darkness is a botnet designed to be used for executing distributed DoS attacks on remote machines. This malware sample starts multiple processes to open communication with the botnet and awaits commands from a remote server. It also uses rootkit-like functionality to prevent detection.

6. PERFORMANCE OVERHEAD

The sensing and detection techniques described in this paper were chosen in part for their relatively low overhead. In this section we present the results of various tests to characterize the performance overhead of our sensors and detector and compare it to the overhead of a commercially available antivirus program. For this comparison, we use Symantec Antivirus Corporate Edition 10 [7].

Symantec Antivirus introduces overhead to the system in a way that differs from our sensor and detection approach. While our approach adds overhead to each system call, Symantec antivirus introduces overhead by performing filesystem and memory scanning tasks in a separate process. The overhead in this case manifests itself as borrowing CPU time from the processes that are performing the benchmark tests. For this benchmark testing, SAV was configured to perform background memory and file system scanning only.

To characterize the overhead introduced to the system, we used the PassMark PerformanceTest benchmark tool [8].

PerformanceTest provides benchmark tests for functions such as 2D and 3D graphics, tests of particular interest because they perform a large number of calls to system functions. For 2D and 3D graphics, we characterize the overhead in terms of frames-per-second (FPS) and vectors-per-second. We also characterize the overhead of the CPU and memory benchmarks in terms of the number of operations they require.

The results of all of these tests are presented in Table 1 in terms of the overhead each application introduces to the system. The results are mixed, showing situations where our techniques both outperformed and underperformed Symantec AntiVirus. As a result, additional testing is desired, which will include tests that are designed specifically to mimic actual system usage, such as software compilation and media encoding and decoding. Furthermore, it is possible to fine-tune the performance of our sensors and detector by adjusting the amount of logging and frequency of data sampling. For this set of tests, all system function accesses were logged and data were sampled at one second intervals.

test	SAV10	our sensors
2d graphics	1.2%	1.0%
3d graphics	0.074%	0.44%
CPU	0.1%	0.5%
memory	1.3%	1.8%

Table 1: Overhead comparison between Symantec Antivirus 10 and our sensors and detector running on a live system

7. MALWARE DETECTION RESULTS

In this section, we present the results of our analysis of the Neyman-Pearson detector described in Equation 5. First, we present the Receiver Operating Characteristic (ROC) of the detector to describe the overall detector performance. In order to get a better sense of how the detector is working, we explore in detail the data collected by a single sensor. To determine the effectiveness of our approach in detecting zero-day malware, we train the detector to detect only one of the seven malware samples, and test the detector on the remaining data sets. Finally, we analyze the detector performance under a very different load scenario and discuss the applicability of this approach to the general malware detection problem.

7.1 Malware detector performance

To characterize the performance of the detector, we first conduct the likelihood ratio test in Equation 5, varying the value of the threshold η. For each threshold value, we perform the likelihood ratio test. We form two data sets to generate the ROC curve: The clean data set contains 16 hours of data gathered from a clean system and the infected data set contains 56 total hours of data, 8 hours for each of the malware samples. For the clean set, we keep track of the number of false positives, and compute the false positive rate α. For the infected set, we compute the true positive rate $(1-\beta)$. By plotting the true positive rate vs. the false positive rate, we obtain the ROC curve of Figure 3. We show the ROC curve for $N = 1, 5$, and 10 samples. The plot shows an increase in detector performance as we increase the number of samples.

We can characterize the performance of a detector in terms of its empirical ROC by computing the area under the curve. The area under the three curves is 0.9823, 0.9937, and 0.9946 for 1, 5 and 10 samples, respectively. Increasing the number of samples beyond 10 results in slight improvement in the area under the curve. Having the ROC allows us to determine for a fixed false alarm rate α, the threshold that enables us to achieve that rate and the corresponding detection rate. For example, if we fix α at 0.02, we can achieve a detection rate of 0.996.

Although we can improve the performance of our detector by taking more samples, in many circumstances we may want an early warning that malware is detected so that appropriate mitigations can be applied, such as isolating the machine from the network, notifying a system administrator, removing the infected files, or shutting down the machine. In practice, if we use the $N = 10$ detector and threshold in Figure 3, we receive a decision from the detector every 10 seconds and can use the successive decision to direct our mitigation procedures.

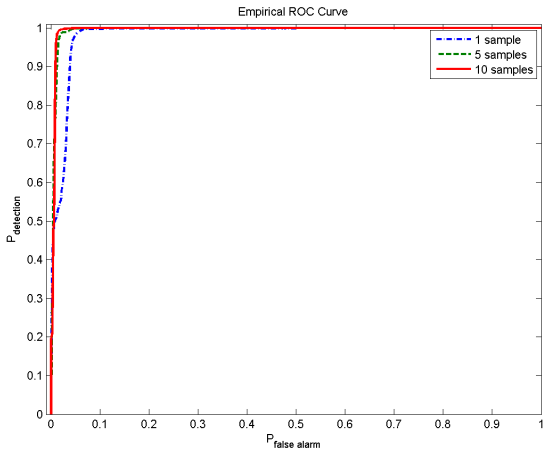

Figure 3: Detector ROC curve for 1, 5 and 10 data samples, showing increase in detector performance for an increase in number of samples used for detection

7.2 Practical malware detector performance

While the ROC curve provides a general overview of the performance of the detector, in practice the detector will be used to classify previously unseen data. As such, we consider the scenario where we use the ROC to choose a threshold for the detector and then test the detector against a separate data set. For this scenario, we assume that we can tolerate a false-positive rate of 2% at $N = 10$ and choose our threshold accordingly. Then, we test the detector against a separate data set, also comprised of 16 hours of clean data and 56 hours of infected data, with each malware sample represented in the data.

For this round of testing, we see a detection rate of 98.9% and a false positive rate of 1.20%, both values comparable those presented in the ROC curve. Thus, given that the system is under a load that is similar to that which was used

for training, the detector is successful in detecting known malware with a low probability of false alarm. The question remains whether this detector is successful in detecting previously unknown malware, but before we explore the answer to that question we will show using an example from one of the sensors why this simple detector is successful in detecting known malware.

7.3 Malware sensor analysis

In the previous section, we have shown our detector to be successful in detecting known malware samples. In this section, we explore the inner-workings of the detector to explain why it is successful. For this analysis, we consider a single sensor, `NtQuerySystemInformation`. `NtQuerySystemInformation()` has many uses, including querying performance counters, getting information about processes running on the system, and getting general system information, such as what memory addresses are available to applications and DLLs. This function is often used by malware authors to gather system information, particularly about applications running on the system. This is done both for monitoring purposes and to protect against detection. For example, Autoit monitors what processes are running on the system to kill those that may be used in its removal, such as the task manager and the command shell.

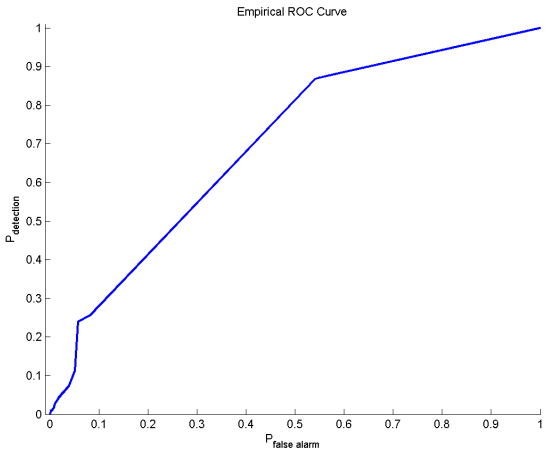

Figure 4: NtQuerySystemInformation ROC curve

If we consider `NtQuerySystemInformation` to be the only sensor we have for our detector, we can obtain the ROC curve of the detector, shown in Figure 4. The ROC shows this detector to be only moderately successful in detecting malware, with an area under the curve of only 0.697. These performance numbers are typical of the highest performing sensors, which include `NtQueryKey`, `NtDelayExecution`, and `NtWriteFile`. From this analysis, it is clear that no single sensor makes an effective classifier. Thus, to achieve the desired performance, we must consider a larger set of sensors, as we do in the detector described by Equation 5.

To see why a single sensor such as `NtQuerySystemInformation` allows us to discriminate between a clean an infected system to a limited extent, we can examine the sensor values in the time domain. Figure 5 shows sensor values over a 5 minute time period for both a clean and infected system. In both cases, we see periodic bursts of calls to `NtQuerySystemInformation()`, and an otherwise constant frequency of calls. However, for the case of the infected system, we see that the constant number of calls made between bursts is higher than the clean system, as the malware regularly uses the function to detect unwanted processes.

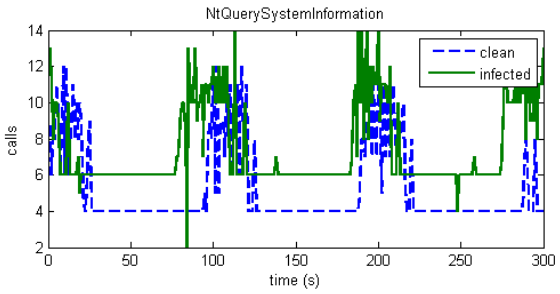

Figure 5: Plot of sensor values for NtQuerySystemInformation for a clean and infected system

7.4 Inoculation against malware infection

In order to test the robustness of this detector in classifying zero-day malware samples, we consider the case when the detector is trained to detect only one of the seven malware samples. This detector is then tested against the data for each of the other six samples. For this testing, we use $N = 10$ samples and train the system on all 16 hours of data for a particular malware sample. Then, we compute the ROC curve for the detector, using 16 hours of clean data to compute the false positive rate. Finally, we choose our threshold such that our false positive rate is fixed at 2%, and use the detector to classify the rest of the samples. For each of these samples, we calculate the the percentage of missed detections, which we present in Table 2.

Analyzing the data in Table 2, we notice that training on Yahlover enables us to detect Autoit, Bohu, and Bybz with a high rate of success. This detector is successful in classifying Autoit and Bybz because they are also behaviorally similar to Yahlover, as they both attempt to propagate over then network. In contrast, we see that we are largely unable to detect the more advanced malware samples Darkness and SpyEye, nor are we successful in detecting Carberp, whose main functionality is stealing data.

If we consider instead the system trained to detect Darkness, we see that we have a high rate of success in detecting all other six malware samples. This indicates that by carefully choosing canonical samples to train our system, we are able to detect a wide array of related malware. These canonical samples should be chosen such that they represent a wide cross section of malicious behavior. In this example, we see that the most sophisticated malware samples, such as Darkness, which contain functionality to hide from detection, infect a machine in multiple ways, and provide remote functionality offer the best detection performance.

Finally, we observe that we can discern what samples of malware are most dissimilar in terms of our statistical model by looking for those pairs that offer poor performance for detecting each other. For example, training for either SpyEye or Yahlover results in poor performance when detecting the other, since Yahlover is a simple worm that propagates using messaging protocols and SpyEye is a sophisticated data stealing tool.

train \ test	Autoit	Bohu	Carberp	SpyEye	Yahlover	Darkness	Bybz
Autoit	—	6.60%	0.52%	0.76%	87.2%	0.76%	25.9%
Bohu	0.00%	—	0.04%	0.04%	0.00%	0.03%	0.00%
Carberp	0.03%	0.21%	—	0.24%	0.14%	0.07%	0.00%
SpyEye	0.00%	0.28%	0.10%	—	13.5%	0.10%	0.07%
Yahlover	0.14%	0.21%	82.5%	84.0%	—	72.3%	0.04%
Darkness	0.00%	0.17%	0.03%	0.10%	0.07%	—	0.00%
Bybz	2.15%	31.6%	23.1%	30.1%	10.0 %	46.6%	—

Table 2: Missed detection percentages when a single malware sample is used to train the system and is used to detect the remaining samples

The results presented in this section show that our sensing and detection techniques require training data gathered from carefully selected canonical samples of malware. If the canonical samples are chosen to represent a wide array of malware functionality, we can inoculate our detector against related malware samples with a low probability of error.

7.5 Detection under different loads

Thus far, we have considered only loads that are characteristic of a web server. In this section, we consider a load that is more closely matched to desktop computing. For this analysis, we used the same testbed as used in the previous section. The only change is that we installed Openoffice.org 3.3.0 and its functional test suite. As load generation for the system, we used the testing tool and tests distributed with OpenOffice. Particularly, we chose the subset of required tests associated with the writer, spreadsheet, drawing, math, and presentation components. Both OpenOffice and the test tool were run on the test machine, and the tests took approximately 4 hours to complete. These tests were performed for three scenarios: a clean system, a system infected with Bybz, and a system infected with Bohu.

Using the model developed with the web server loads, we are unable to classify the data gathered under the desktop-like OpenOffice load. Namely, we experience a near-100% false positive rate. Instead, we considered a situation where we train the detector using the desktop load, and then attempt classification. We plot the ROC curve of this detector in Figure 6. This ROC curve shows that although we cannot use the model from the server load to perform malware detection, retraining our models on the desktop load offers performance comparable to the server load with $N = 10$. This result indicates that if the model of a clean and infected system are developed for the specific type of load the system will be experiencing, we can reasonably expect good performance of this Neyman-Pearson detector.

We also consider the case where the model is built using both types of load. For this test we used both the 12 hours of data from desktop load and 12 hours of data from the server load to avoid biasing the resulting model. The ROC of this detector is shown in Figure 7. This figure shows that even when increasing the sample size to $N = 30$, we are still unable to achieve the performance of the detector in either of the scenarios where we considered only one type of load. A closer analysis of the individual sensor data indicates that by training on different types of load, we have introduced additional complexity to the marginal distribution of each sensor, such that a geometric distribution is no longer a good approximation. This has caused the performance of the overall detector to degrade.

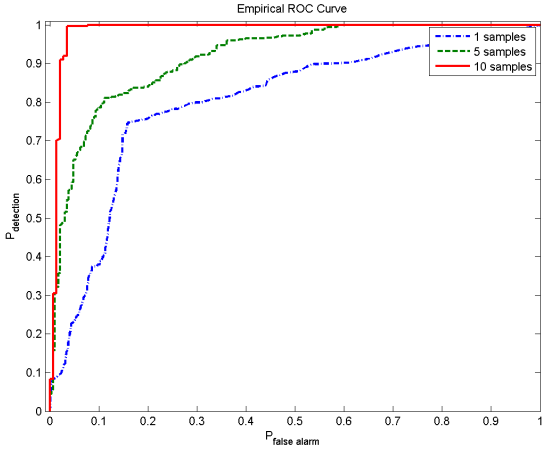

Figure 6: ROC curve for system under desktop load, showing similar performance to server load ROC curve

The results of this section provide us with two conclusions: First, if the load used for training is sufficiently similar to the actual load of the system, we can expect good performance for the detection of zero-day malware. However, if the load differs from the training load, we cannot expect the detector to be effective. Second, while the Neyman-Pearson test with independence assumptions and geometric marginal distributions provides an adequate model under specific types of load, the model breaks down when considering diverse loads simultaneously, and causes the performance of the detector to degrade.

8. THREATS TO VALIDITY

The first threat to validity is that we have restricted the types of load placed on the system in such a way that we have made the malware more easy to detect. We presented results for a server-like and desktop-like load, and showed that training on one does not allow us to detect under the other, and that training on both results in decreased detector performance from the case where we train on only one of the loads. However, our results are promising in showing that even under a fundamentally different type of load, the detector still provides adequate discrimination between

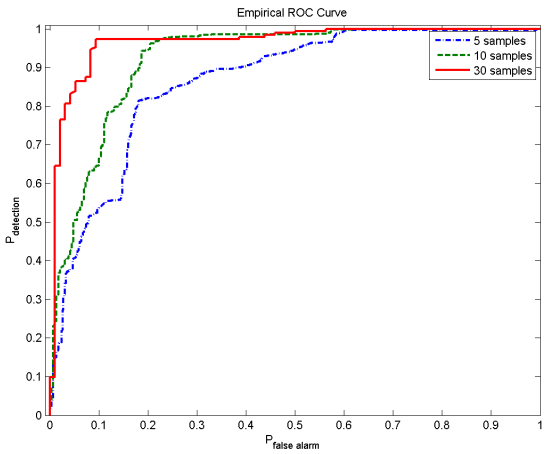

Figure 7: ROC curve for system under both desktop and web server loads, showing degraded performance

clean and infected systems. Thus, these results highlight the need for a more accurate statistical model that can more accurately model the sensor values under diverse loads, which is af subject of our future work.

Next, we have assumed our sensors to be independent when our analysis of the sensor data showed certain sensors to be correlated. While we chose to assume independence for mathematical and computational simplicity, previous empirical research has shown naive Bayesian detectors operating under strong independence assumptions to be effective in classifying many real-world data sets for which independence does not hold true, including some applications in malware detection. This naive approach serves as an excellent baseline to compare to more complex detectors, which we plan to do as part of our future work.

Finally, our sensor-detector system operates on the operating system where we are performing detection, leaving our system vulnerable to attack from malware. The user-mode controller is particularly vulnerable to attack since it is not afforded the limited protections that executing at the operating system level provides. This, however, does not contrast with how commercially available antivirus detectors function: they are comprised of a user-level interface and system-level services for detection. Thus the vulnerability of our sensor-detector system is not unique, but is common to malware detectors that run on the same operating system as the malware they are detecting. For example, our sensors are vulnerable to attacks by rootkits that hook system calls to enable their stealth functionality. Like other malware detectors, if these sensors are deployed on a live system, we must take actions to mitigate the risk of attack such as monitoring whether the system calls remain hooked by our sensors and are returning valid data.

9. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach to malware detection using a suite of dynamic sensors and a Neyman-Pearson detector. We have implemented the sensor suite as a kernel-level device driver for Microsoft Windows XP and implemented a Neyman-Pearson test to classify the data. We performed a case study on the detection of seven malware samples and reported results that demonstrate the effectiveness of this approach in detecting malware under two distinct types of load.

We have shown the sensors and detection technique to be effective in classifying systems as either clean or infected with a high probability of detection and low probability of false alarm. Furthermore, we have shown that our approach is able to detect unknown malware, provided the system has been trained to detect similar malware. By using canonical examples of malware to train our detector, we inoculate the detector against similar types of malware to detect zero-day attacks.

Finally, we have presented a sensing technique that is both low-overhead and effective in capturing the differences between a clean and infected system. Furthermore, we have presented a technique for classifying the data gathered from these sensors that is both mathematically simple and computationally inexpensive.

This paper represents our first approach to malware detection using the described sensor suite. Accordingly, we plan to continue our analyses of the overhead introduced by the sensors and detector and compare it to the overhead of commercially available detectors.

We plan to expand on the detection technique presented in this paper to address the assumption of independence. This work will include feature reduction to choose the set of independent sensors to model the system and the development of a more accurate statistical model to describe the sensor distributions. We also plan to perform comparisons between our detection-theory based approach to malware detection and other common approaches, including ones from machine learning and computational geometry.

Finally, we plan to continue to collect data using our sensor suite for different types of data loads and different malware samples. We will use these expanded data sets to test and refine our detection technique and to expand it to the problem of classifying different types of malware.

10. REFERENCES

[1] Apache, 2010. http://httpd.apache.org.
[2] Drupal, 2010. http://www.drupal.org.
[3] Microsoft, 2010. http://www.microsoft.com.
[4] Mysql, 2010. http://www.mysql.com.
[5] Php, 2010. http://www.php.net.
[6] Silverstripe, 2010. http://www.silverstripe.org.
[7] Symantec, 2010. http://www.symantec.com.
[8] Passmark software, 2011. http://www.passmark.com/.
[9] Y. Al-Nashif, A. Kumar, S. Hariri, G. Qu, Y. Luo, and F. Szidarovsky. Multi-level intrusion detection system (ml-ids). In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 131 –140, 2008.
[10] A. Baker and J. Lozano. *The Windows 2000 Device Driver Book*. Prentice Hall, 2nd edition, 2000.
[11] P. Dabak, S. Phadke, and M. Borate. *Undocumented Windows NT*. Hungry Minds, Oct. 1999.
[12] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[13] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pages 201 –203, 2010.

[14] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley Professional, 2005.

[15] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[16] B. Krebs. White house ecard dupes dot-gov geeks, Jan. 2011.

[17] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 251–261, New York, NY, USA, 2003. ACM.

[18] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 399–412, New York, NY, USA, 2010. ACM.

[19] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441, Dec. 2007.

[20] R. Moskovitch, Y. Elovici, and L. Rokach. Detection of unknown computer worms based on behavioral classification of the host. *Comput. Stat. Data Anal.*, 52:4544–4566, May 2008.

[21] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[22] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):1–54, 2008.

[23] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[25] M. Russinovich and B. Cogswell. Windows nt system-call hooking, Jan 1997. http://www.drdobbs.com/184410109.

[26] K. Schreiner. New viruses up the stakes on old tricks. *Internet Computing, IEEE*, 6(4):9 – 10, 2002.

[27] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 38, Washington, DC, USA, 2001. IEEE Computer Society.

[28] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *ID'99: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring*, pages 4–4, Berkeley, CA, USA, 1999. USENIX Association.

[29] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, pages 1–30, 2011. 10.1007/s10844-010-0148-x.

[30] E. Stehle, K. Lynch, M. Shevertalov, C. Rorres, and S. Mancoridis. On the use of computational geometry to detect software faults at runtime. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 109–118, New York, NY, USA, 2010. ACM.

[31] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, Feb. 2005.

[32] H. L. VanTrees. *Detection, Estimation, and Modulation Theory*. John Wiley and Sons, 2001.

[33] C. Wang, J. Pang, R. Zhao, and X. Liu. Using api sequence and bayes algorithm to detect suspicious behavior. In *Communication Software and Networks, 2009. ICCSN '09. International Conference on*, pages 544 –548, 2009.

[34] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. pages 203–222, 2004.

[35] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[36] T. Yong and C. Shigang. An automated signature-based approach against polymorphic internet worms. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7):879 –892, 2007.