Multi-Channel Change-Point Malware Detection

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

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

Abstract—**The complex computing systems employed by governments, corporations, and other institutions are frequently targeted by cyber-attacks designed for espionage and sabotage. The malicious software used in such attacks are typically custom-designed or obfuscated to avoid detection by traditional antivirus software. Our goal is to create a malware detection system that can quickly and accurately detect such otherwise difficult-to-detect malware. We pose the problem of malware detection as a multi-channel change-point detection problem, wherein the goal is to identify the point in time when a system changes from a known clean state to an infected state. We present a host-based malware detection system designed to run at the hypervisor level, monitoring hypervisor and guest operating system sensors and sequentially determining whether the host is infected. We present a case study wherein the detection system is used to detect various types of malware on an active web server under heavy computational load.**

I. Introduction

Our primary defense against malicious software (malware) has traditionally been antivirus software, which use static signature-based detection techniques to identify potential malware. Popular due to their low false-alarm rates and ease of use, antivirus software require new malware samples to be discovered and analyzed before they can be detected, leaving hosts vulnerable to new malware during the time period between the sample first being used in a cyber-attack and the creation of detection signatures for that sample [1]. The complex, ultra large-scale systems (ULS) used by governments, corporations, and other institutions, are particularly vulnerable to new malware, since these systems are constantly subject to cyber-attacks and their size and complexity complicate detection [2].

Antivirus software are also ineffective at detecting obfuscated variants of known malware [3], [4]. Obfuscations are applied to malware using specialized software that reorders, encrypts, compresses, recompiles, or otherwise changes the code without altering its function [5]. Obfuscations can also be applied automatically and incrementally, as is the case with metamorphic and polymorphic malware that mutate as they propagate [6]. Obfuscating malware to evade detection is now common practice since the engineering effort required to design new malware far exceeds the effort to obfuscate existing malware. Accordingly, the majority of new antivirus detection signatures are not created for new malware, but rather for obfuscated variants of known malware [7].

In this paper, we present a malware detection system designed to detect obfuscated variants of known malware and previously unseen malware that are behaviorally similar to known malware. The malware detection system monitors data from a suite of sensors installed on a host server at both the operating system and hypervisor levels, and processes the sensor data sequentially as they become available, using the data to infer whether the host is executing malware.

We pose the malware detection problem as a change-point detection problem [8], wherein the goal is to detect whether a host is infected with malware by detecting changes in distribution of the sensor data as quickly as possible. We assume that the host being monitored is initially clean and free of malware and that during this period of time we are able to establish a baseline of normal operation for the host. Assuming that the host may become infected with malware at any time, our goal is to determine whether the host is infected so that appropriate mitigative actions can be performed to limit data loss, data theft, further propagation of the malware, and disruption of services.

We present a malware detection system that treats the malware detection problem as a multi-channel, decentralized detection problem. The problem is multi-channel because each sensor measures a unique phenomenon and reports data that are governed by a distinct probability distribution. The problem is decentralized because detection is performed at the sensor level, wherein each local detector uses data from only one sensor to infer whether the host is infected. The global decision is made by a data fusion center (DFC), which sequentially processes the decisions from the local detectors to infer whether the host is infected.

Finally, we present a case study using the described malware detection system on a virtual machine host running a web server under heavy computational load. During testing, the host is originally clean and becomes infected with malware at a randomly selected time instance. Two hundred different malware samples, all gathered from the wild in the past year, are used for the study. We examine the effectiveness of the detection system both in terms of its overall detection accuracy and its average time to detection.

The remainder of this paper is organized as follows: After a description of related research in the following section, the malware detection problem is formally stated in Section III and details of the detection system are provided in Section IV.

Section V provides a detailed description of the case study and analysis of the detection results. Section VI concludes our analysis and provides an overview of our ongoing work.

II. Related Work

This work draws on a substantial body of prior research in the areas of malware detection and intrusion detection. Early work in malware detection focused on static detectors which, like antivirus software, label static files as either clean or infected [9], [10]. This early work identified useful features for malware detection, such as instruction sequences, strings, and library calls. However, these features have been shown to be difficult to extract from obfuscated malware, motivating alternative approaches to the malware detection problem that do not rely on static analysis [11].

Dynamic malware detectors are designed to detect whether malware are executing on live a host. Early work on dynamic detectors demonstrated system call sequences to be useful features for malware and intrusion detection, using a database of known benign call sequences to perform anomaly detection on kernel traces [12]. More recently, this work has been expanded to show that system call sequences together with their input arguments can be used for malware classification [13]. Similar work using system calls and clustering algorithms for malware detection in a virtualized environment has yielded promising results [14]. Work by Landi *et al.* demonstrated that call sequences alone are not very good indicators of malicious behavior, instead using abstracted descriptions of system activity [15].

Taint analysis techniques, which monitor data introduced to a host, have been shown to be effective in detecting malware, especially privacy-breaching malware [16], [17], [18]. However, some research has shown that computational complexity limits taint analysis to simple systems and offline analysis [19], [20]. Other approaches to malware detection have included monitoring system performance metrics [21], [22], virtual machine sensors [23], and a combination of static and dynamic features [24]. For a more detailed survey of malware detection techniques, the interested reader is referred to a survey by Idika and Mathur [25] and an evaluation of commercial antivirus software by Sukwong *et al.* [3].

In the related field of network intrusion detection, sequential detection techniques applied to network-based features have been demonstrated to provide an accurate means of detecting network intrusions [26] and denial of service attacks [27]. Hidden Markov models have been shown to be an effective tool for detecting cyber-attacks using system call traces [28].

The malware detection system described in this paper bears similarity to previous work in that it is a dynamic detection system that monitors features at the operating-system level [21] and the hypervisor-level [23] to infer the execution of malware. The described system is unique in its decentralized application of two-sided sequential detection techniques described Page [29] for malware detection and its use of a data fusion center for global decision making. The merits of the described system that set it apart from the previous work include the low computational complexity of the detection system, which allows for real-time detection of malware infection on a live host, and the sequential formulation of the malware detection problem that focuses on detecting malware quickly and accurately.

III. The Malware Detection Problem

The present work focuses on the detection of clandestine malware designed for espionage and sabotage that do not present obvious signs of infection. Clandestine malware are designed to be difficult to detect, analyze, and remove. Such malware use a variety of tricks to avoid detection and removal, including running as background processes, system services, or device drivers; disguising themselves as legitimate software; and altering the host OS's security configuration. Furthermore, malware are often protected against various types of reverse engineering, including interactive debugging and disassembly [1].

In order to perform the malicious tasks for which they are designed, malware must interact with the host OS. Furthermore, the tricks used by the malware to thwart detection, analysis, and removal also require OS interaction. We assume, based on previous research in behavioral malware detection (see Section II), that such OS interaction causes perturbations in a set of observable features, and by monitoring these features we can infer whether a host is executing malware. For this paper, we focus on features measured by software sensors at the guest OS level and the hypervisor level.

We pose the malware infection in the following way: We assume that when a host is initially configured, it is clean, *i.e.,* there is no malware executing on the host. Furthermore, we assume that we can monitor the host when it is initially configured to establish a baseline model of the clean host. Due to security vulnerabilities, intrusions, *etc.*, the host can become infected with malware at any time after the initial model is established. We say a host is infected if there is malware actively executing on the host.

We define the infection time t_i as the time at which the malware begins execution. Our goal is to detect that the host is infected as close to time t_i as possible so mitigative action can be taken to prevent data theft, data loss, system down time, further propagation of the malware, and other undesirable effects. We define the time that elapses between the infection time and the time we detect the malware as the detection delay t_d. We assume that t_i is not guaranteed to be finite and that the distribution of t_i is not known *a priori*, *i.e.,* we do not know if or when a particular host might become infected with malware. Furthermore, we also do not know *a priori* how likely it is that a particular host might become infected with malware [30].

For a clean system, the feature data for each sensor are considered to be a sequence of independent random variables, where the data for the m^{th} feature are given by $\mathbf{x}_m = \{x_{m,1}, x_{m,2}, ..., x_{m,t_i-1}\}$ distributed according to a probability density $p_{\Theta_{m,0}}(\mathbf{x}_m)$, where $\Theta_{m,0}$ is a vector of the parameters of the distribution.

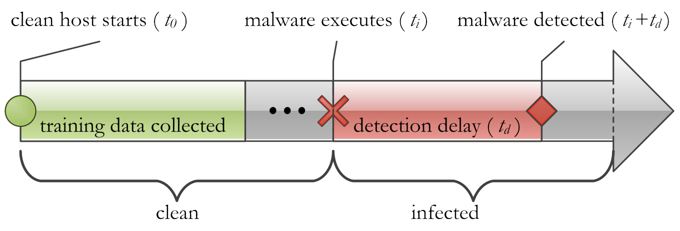
clean host starts (t_0)
malware executes (t_i)
malware detected (t_i+t_d)
training data collected
detection delay (t_d)
clean
infected

Fig. 1. Malware detection timeline

At time t_i the malware executes, and the feature data form a new sequence of independent random variables $\mathbf{x}_m = \{x_{m,t_i}, x_{m,t_i+1}, x_{m,t_i+2}, ...\}$ distributed according to a probability density $p_{\Theta_{m,1}}(\mathbf{x}_m)$ and $\Theta_{m,1} \neq \Theta_{m,0}$. The goal is to determine if and when the distribution of the feature data changes from $p_{\Theta_{m,0}}(x_m)$ to $p_{\Theta_{m,1}}(x_m)$ for each sensor, where $\Theta_{m,0}$ and $\Theta_{m,1}$ are uniquely determined for each sensor. This type of detection problem, detecting a sudden change in distribution, is referred to as quickest detection, change detection, or change-point detection [8], [31], [32].

Finally, we assume that at infection time t_i a subset of the M total features will experience a change in distribution, and by monitoring which subset of sensors experience a change in distribution, we can infer the presence of malware on a host. The subset of features that change distribution will not necessarily be the same for all malware samples but rather will be determined by the function and design of the malware.

To summarize, the goal is to detect the time t_i when a system changes from a clean to infected state, under the assumption that at time t_i a subset of the features will change distribution from $p_{\Theta_{m,0}}(x_m)$ to $p_{\Theta_{m,1}}(x_m)$, where $\Theta_{m,0}$ and $\Theta_{m,1}$ are uniquely determined parameters for each feature distribution. We assume that $\Theta_{m,0}$ can be learned from a limited set of clean training data and that $\Theta_{m,1}$ is unknown. A depiction of the timeline of these events is provided in Figure 1, which shows the host starting, the clean training data being collected, the malware executing at time t_i, and the malware being detected after a brief delay t_d.

IV. Detection System Design

The malware detection system (MDS) is designed to detect whether the guest OS running inside a virtual machine (VM) is infected with malware. We assume that the VM is managed by a hypervisor or a virtual machine monitor (VMM). Figure 2 shows the architecture of the MDS, comprising four major components:

1) The **sensors**, which monitor the host at the hypervisor and guest OS levels;
2) The **feature extractor**, which samples the feature data from the sensors;
3) The **local detectors**, which perform detection on each stream of feature data independently; and
4) The **data fusion center** (DFC), which uses the decisions from the local detectors to infer whether the guest OS is infected.

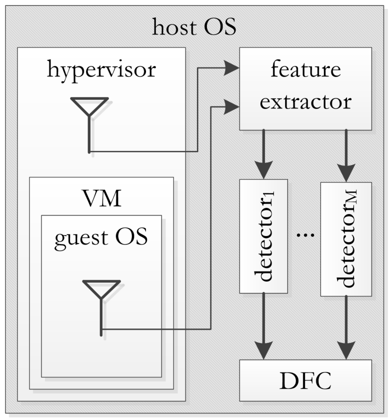

Fig. 2. Malware detection system architecture

A. Sensors and feature extractor

The MDS is designed to work with any set of numeric features collected from applications, the guest OS, the hypervisor, or the host OS. In its current configuration, the features used by the MDS are data sampled from sensors in both the guest OS and the hypervisor. The features fall roughly into the following categories:

- Memory usage,
- Processor usage,
- Disk usage,
- Page file usage,
- Hardware usage, and
- Network usage.

To mitigate the risk of malware running on the guest OS tampering with the MDS, the hypervisor sensors, feature extractor, local detectors, and DFC are all located outside the guest OS, as shown in Figure 2. The guest OS sensors are built-in to the kernel of the guest OS, making them more robust than, for example, sensors running as a user-level process or sensors hooking the System Service Dispatch Table (SSDT) that can be easily modified [33]. Furthermore, the layered approach of using sensors at both the guest OS and hypervisor is designed to reduce the likelihood that compromised sensors at a particular layer would result in missed detections.

B. Local sequential detectors

The feature extractor sends streams of feature data to a series of local detectors, each of which independently infers whether the OS is infected based only on a single feature. This detection architecture was chosen because the subset of features exhibiting a change in distribution when malware are executed differ for each malware sample, likely due to the differing function and implementation of the malware.

Since we have posed the malware detection problem as a change-point detection problem, the local sequential detectors each perform an implementation of a popular change-point detection algorithm called Page's cumulative sum (CUSUM) test [29]. The algorithm, originally described as a method for detecting faults in industrial production processes, can be implemented as a repeated cumulative log-likelihood ratio test with an adaptive detection threshold.

The CUSUM algorithm was originally formulated as a method to detect a change in a scalar parameter of a distribution from a known value θ_0 to another known value θ_1. Thus, we assume that the feature data can be approximately described by a known parametric distribution both before and after the change. However, we do *not* assume that the value of the parameter is known after the change. Rather, we assume that the change will have a magnitude of at least δ, *i.e.*:

$$|\theta_0 - \theta_1| \geq \delta. \tag{1}$$

Thus, we use a double-sided implementation of Page's CUSUM algorithm to detect whether each of the parameters of the distribution exhibits either an increase or decrease of magnitude δ, uniquely determining δ for each parameter.

While the CUSUM algorithm can be used on any parametric distribution, we observed during testing that for the majority of the features, the data can be approximately described using a normal distribution. We formally define the CUSUM test performed at each local detector as the detection of a change in either the mean or the variance of the feature data. Thus, we have four possible changes at each sensor. Either:

1) The mean increases from μ_0 to $\mu_0 + \delta_\mu$,
2) The mean decreases from μ_0 to $\mu_0 - \delta_\mu$,
3) The variance increases from σ_0^2 to $\sigma_0^2 + \delta_{\sigma^2}$, or
4) The variance decreases from σ_0^2 to $\sigma_0^2 - \delta_{\sigma^2}$.

We begin by defining the normal probability distribution function:

$$p_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{2}$$

Next, we define the set of four cumulative log likelihood ratios (LLR) that are computed by each sensor. We define these four cumulative LLR as $g_n^{(1)}$, $g_n^{(2)}$, $g_n^{(3)}$, and $g_n^{(4)}$, where the subscript indicates that it is the LLR at the n^{th} data sample and the superscript indicates which of the four changes in the list above it is used to detect. We define the cumulative LLR recursively as:

$$\begin{aligned} g_n^{(1)} &= g_{n-1}^{(1)} + \log\left(\frac{p_{\mu_0+\delta_\mu,\sigma_0^2}(x)}{p_{\mu_0,\sigma_0^2}(x)}\right) \\ g_n^{(2)} &= g_{n-1}^{(2)} + \log\left(\frac{p_{\mu_0-\delta_\mu,\sigma_0^2}(x)}{p_{\mu_0,\sigma_0^2}(x)}\right) \\ g_n^{(3)} &= g_{n-1}^{(3)} + \log\left(\frac{p_{\mu_0,\sigma_0^2+\delta_{\sigma^2}}(x)}{p_{\mu_0,\sigma_0^2}(x)}\right) \\ g_n^{(4)} &= g_{n-1}^{(4)} + \log\left(\frac{p_{\mu_0,\sigma_0^2-\delta_{\sigma^2}}(x)}{p_{\mu_0,\sigma_0^2}(x)}\right). \end{aligned} \tag{3}$$

Each cumulative LLR above will decrease if no change in parameter occurs and will increase if the corresponding change in parameter occurs. Since our goal is to detect only an increase in the LLR associated with a change in parameter, we reset the LLR to zero each time it becomes negative. This reset has the effect of transforming the standard log likelihood ratio test (LLRT), where the values in Equation 3 are compared

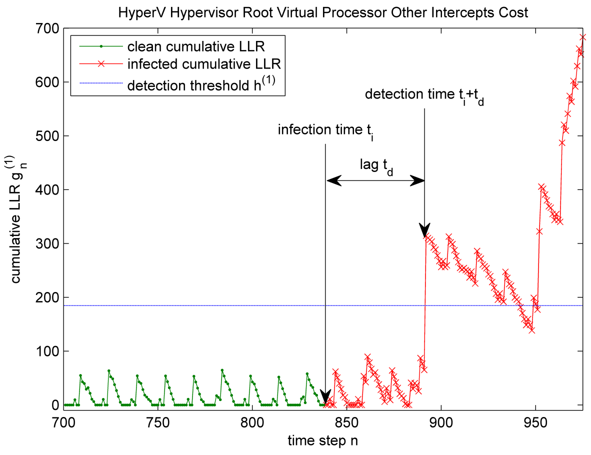

Fig. 3. Page's CUSUM test

to a fixed threshold, into an adaptive threshold test. We modify the LLR in Equation 3 as follows, where $a \in \{1, 2, 3, 4\}$:

$$g_n^{(a)} = \begin{cases} g_n^{(a)} & \text{if } g_n^{(a)} > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Next, a modified LLRT is performed on each of the four transformed LLR described by Equations 3 and 4 by comparing each ratio to a threshold $h^{(a)}$ in order to make a decision $d_n^{(a)}$:

$$d_n^{(a)} = \begin{cases} 1 & \text{if } g_n^{(a)} > h^{(a)} \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

The effect of resetting the cumulative LLR and using a fixed threshold as shown in Equations 4 and 5 is that $g_n^{(a)}$ will remain near zero so long as a change in the distribution of the data has not occurred, and will increase after the corresponding change has occurred.

Figure 3 provides a visual depiction of Page's CUSUM test being used to detect an increase in the mean of a particular feature, the virtual processor other intercepts cost, as reported by the hypervisor. Before the malware infection occurs, the cumulative LLR values periodically spike above zero and are frequently reset to zero when the LLR becomes negative. At $t_i = 840$ seconds, the malware is executed and the system is infected. After that time, we note an increase in the LLR $g_n^{(1)}$. Once it crosses the detection threshold $h^{(1)}$, the local detector reports the decision that malware has been detected.

Each LLRT results in a new decision $d_n^{(a)}$ after each new data point is considered. Whenever the decision is zero, indicating no malware has been detected, the test continues when the next data point arrives. Whenever the decision is one, indicating malware has been detected, the pair of tests for the parameter in which the change was detected stops since a decision has been reached, and the local detector notifies the DFC that malware has been detected. If a change is detected in both parameters, all four tests stop, and the local detector notifies the DFC that malware was detected in both parameters.

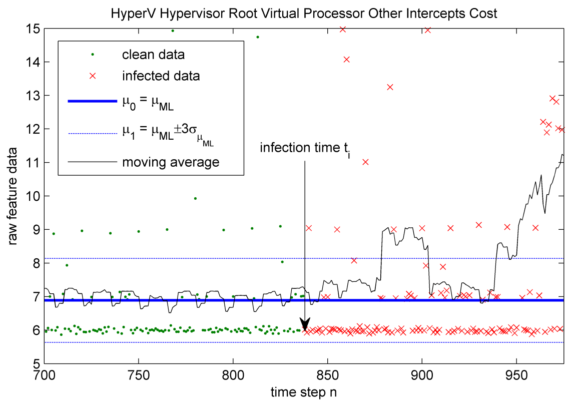

Fig. 4. Feature data change in parameter

Once the tests have stopped, subsequent data are discarded until the DFC notifies the detector that it should continue testing.

The local detection process of computing four adaptive, cumulative LLR and comparing them to a threshold to make a decision is performed independently for each of the M features. Because each sensor measures a different phenomenon, the parameters of the distributions and detection thresholds are uniquely determined for each feature.

The parameters μ_0 and σ_0^2 are determined during an initial training period, when the host is in a clean state and operating normally, as shown in the timeline in Figure 1. During the training period, data are collected and separated into blocks of equal length and the maximum likelihood estimates of the parameters are computed. The standard deviations of the estimates are used to determine the magnitudes δ_μ and δ_{σ^2} of the changes to be detected.

If the maximum likelihood estimate of the mean is given by μ_{ML} and has a standard deviation given by $\sigma_{\mu_{\text{ML}}}$, then the goal will be to detect a change of magnitude $\delta_\mu = n\sigma_{\mu_{\text{ML}}}$, where n is a multiplier which indicates how many standard deviations of change we aim to detect. δ_{σ^2} is similarly determined using the maximum likelihood estimate of the sample variance and the standard deviation of the estimate.

For example, Figure 4 shows the virtual processor other intercepts cost data used in Figure 3. The bold horizontal line indicates μ_{ML}, the maximum likelihood estimate of μ_0, and the parallel lines indicate $\mu_0 \pm 3\sigma_{\mu_{\text{ML}}}$, the parameters after the change used for computing the LLR $g_n^{(1)}$ and $g_n^{(2)}$. Figure 4 also shows a plot of the raw feature data and its moving average. Before t_i, the moving average closely matches $\sigma_{\mu_{\text{ML}}}$, and after t_i, the moving average increases, resulting in the increase in $g_n^{(1)}$ and the positive detection indicated in Figure 3.

Finally, the detection thresholds $h^{(k)}$ are set to be a fixed number of standard deviations above the mean $g_n^{(a)}$ value observed during the training period. The appropriate number of standard deviations is determined experimentally as described in Section V.

The selected approach described in this section is asymptotically efficient but sub-optimal, as the thresholds are selected in an *ad hoc* manner based on previously detected infections. The optimal selection of thresholds for decentralized quickest detection is known to require simultaneous solution of coupled algebraic equations [34].

C. Data fusion center

The data fusion center (DFC) is responsible for making the global decision as to whether the host is infected. The DFC receives, at each iteration, reports from each local detector indicating whether a change was detected in one or both of the parameters and uses a fusion rule to arrive at a global decision. The DFC employs a k out of N fusion rule, where k is the threshold of the number of positive detections and N is the total number of decisions reported by the local detectors. Because each local detector detects changes in two different parameters, N is given as $2M$ where M is the number of local detectors. Thus, the DFC decides that the host is infected if at least k out of the N decisions reported by the local sensors are positive.

One phenomenon observed during testing is that some sensors report transient data spikes when running on a clean host. The magnitude of the spikes cause an increase in both the mean and the variance of the data, causing the corresponding local detectors to report a positive detection. Over time, spikes from multiple sensors yield multiple false positive detections leading eventually to a global decision that the system is infected even though it is in fact clean, *i.e.* a global false positive. Our testing indicated that these transient spikes in the sensor data are isolated events, happening at only one or two sensors at a time. This is a sharp contrast to the behavior exhibited when malware infection occurs, characterized by near-simultaneous changes in distribution at multiple sensors, leading to true positive detections from multiple local detectors in quick succession.

To mitigate the occurrence of global false positives due to transient data spikes, the DFC tracks not only the decisions made by each of the local sensors, but also the times at which the decisions are made. If the DFC fails to make a global decision that the host is infected in a fixed time window after a local detector indicates a positive detection, the DFC notifies the local detector to resume normal operation and reset its decision to zero. This modification to the k out of N decision rule used by the DFC reduces the number of global false alarms reported by the DFC, especially during prolonged periods of clean operation of the host. The time window is determined experimentally as discussed in Section V.

V. Case Study

To assess the usefulness of the described malware detection system (MDS), we performed a case study using the MDS to detect whether a host under heavy computational load was infected with one of 200 different malware samples collected from the wild. The case study was performed on a custom-built malware detection testbed.

Malware often require a network connection to perform various tasks, such as "phoning home" to indicate that a new host has been infected, sending out spam email messages, sending stolen information to a remote host, attempting to propagate to other hosts, or communicating with other compromised hosts. Accordingly, we designed a testbed that would allow malware to establish connections to a remote host and have that remote host provide some limited interaction with the malware. The testbed comprises two physical machines: a server that becomes intermittently infected with malware and a network emulator.

The network emulator is the portion of the testbed designed to interact with malware that attempt to communicate over the network. The network emulator runs two major pieces of software: (1) a DNS server and (2) the Dionaea low interaction honeypot [35]. The DNS server is used as the primary DNS server for the testbed and resolves all hostnames to the IP address of the honeypot. Furthermore, the network emulator is also configured as the default gateway for all traffic originating on the testbed, forwarding all traffic to the honeypot. The honeypot accepts incoming connections using many different protocols, including SMB, http, ftp, and MySQL, and interacts with the malware in order to gain copies of their malicious payloads. Thus, whenever malware attempt to communicate over the network, they are forwarded to the honeypot, which provides limited network interaction.

The server that becomes intermittently infected with malware runs Microsoft Windows Server 2012 and Microsoft's Hyper-V hypervisor-based server virtualization software [36]. The virtual machines that are infected with malware run Microsoft Windows 7, Microsoft IIS Express Web Server, and Microsoft SQL Server [36], and are configured to host a default installation of the Drupal Content Management System [37].

In order to characterize how the MDS will perform when deployed on a live system, the testbed is designed to automatically and repeatedly run all of the tests in the Drupal test suite in a random order determined each time the virtual machine is started. The Drupal test suite consists of 66 different categories of tests, each exercising different Drupal functions, such as file access, database access, image processing, PHP functions, content searches, system updates, content creation, and content access. All of the tests are used to ensure heterogeneity in the intensity of the load placed on the host and heterogeneity in the types of operations being performed. The test order is randomized to ensure the load during the training period varies and that the malware infection time t_i does not always coincide with a particular test.

The final component of the testbed is the mechanism that infects the VM with malware. Each time the VM is started, the MDS automatically starts, first training the detectors using the initial data retrieved from the sensors and then performing detection. A separate daemon is used to randomly determine when the malware is executed on the VM. For the case study, the daemon executes the malware anywhere between 15 minutes and 2 hours after the VM is started, to ensure that the malware executes independently of the underlying test load.

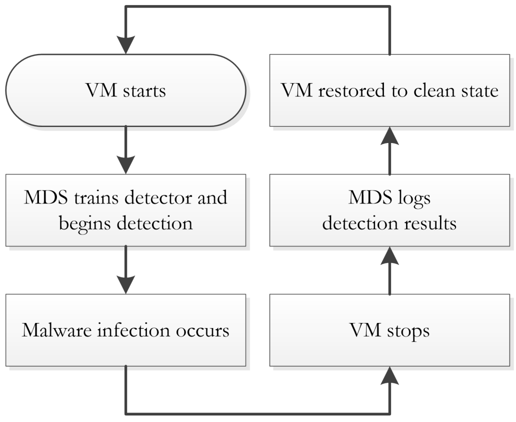
VM starts
VM restored to clean state
MDS trains detector and begins detection
MDS logs detection results
Malware infection occurs
VM stops

Fig. 5. Flow chart of malware testbed execution

Figure 5 shows a flow chart summary of the testing process. When the VM starts, the test load begins immediately, and the MDS trains the detector and begins detection. The VM executes in a clean state for a randomly determined period of time until t_i, when the malware executes and the system enters and infected state. The VM continues to run in an infected state for ten minutes, while the MDS gathers data and performs detection. This ten minute period is used for testing only. In an actual deployment, the MDS would apply mitigation as soon as the malware are detected. Finally, the VM is stopped, the MDS saves all of the data it collected, the VM is restored to its clean state, and the next test cycle begins with another randomly chosen malware sample, permutation of the background load, and infection time t_i.

A. Malware

The malware used in the case study come from two different sources. First, a custom-built program is used to crawl blacklisted sites known to be hosting malware and download any samples it finds. Second, the Dionaea honeypot used in our network emulator is also used for its intended purpose as a honeypot, which interacts with infected hosts on the Internet to obtain malware samples. All of the malware are scanned and classified using the VirusTotal free online virus scanner [38], which scans the malware using a variety of different commercially available antivirus software.

For the case study, we selected 200 malware samples at random from our malware collection. Of those malware samples, 16 were first seen between 2006 and 2011, and the remaining samples were first seen in 2012. Although there is no universal malware taxonomy, we can use malware detection signatures to approximately determine the classes of the selected malware. Using the Microsoft Malware Protection Center (MMPC) [36] naming scheme, we can assign the malware into the categories listed in Table I. The categories are defined as follows:

- **Backdoors** are malware that provide clandestine remote access;

TABLE I
MALWARE SAMPLE CLASSIFICATION

Category	Count
Backdoor	27
PWS	45
Trojan	69
VirTool	19
Virus	8
Worm	16
Undetected	8
Other	8

- **PWS** are password stealers, malware that steal personal information;
- **Trojans** are malware that are disguised as legitimate software;
- **Viruses** are malware that replicate automatically;
- **Worms** are malware that propagate automatically over a network; and
- **VirTools** are tools such as rootkits that provide clandestine access to a host.

In addition to the above categories, two other categories are listed in Table I. **Other** refers to malware that do not fall into any one of the above categories and **undetected** refers to malware that were not detected by any MMPC signatures as of January 2013. These eight undetected malware samples are indicative of the shortcoming of signature-based antivirus detection: although these samples were first seen as early as May 2012, virus detection signatures were still not available up to seven months after their discovery.

Several of the more commonly known malware families are represented in the 200 malware samples used for testing, including Bifrose, Fynloski, Kelihos, ZBot, Swisyn, and Ramnit. The samples also include multiple variants within some of these malware families. Before they were used for testing, we analyzed kernal traces and network logs to verify that the malware performed some observable malicious task, such as attempting to propagate over the network, altering the host configuration, or installing additional software.

B. Feature selection

In its current configuration, 667 sensors exported by the guest OS and 989 sensors exported by the hypervisor are available for detection. Rather than use all $1,656$ available sensors for detection, we first performed feature selection using a randomly selected subset of 20% of the malware samples to determine which features are most useful for malware detection.

The motivation for performing feature selection is two-fold: First, by reducing the total number of features used for detection we reduce both the computational requirements of the detectors and the computational overhead introduced by the sensors. Second, experimentation revealed that only a subset of the features exhibited detectable changes in distribution that were correlated with the introduction of malware to a host. Other features exhibited changes in distribution that were uncorrelated with the introduction of malware, likely due to the changes in the background load running on the testbed. The inclusion of such features in the detector would result in spurious detections leading to decrease in the overall accuracy of the MDS. Such features are deemed not useful for malware detection and are removed from consideration during the feature selection step.

The feature selection process proceeds in three stages. First, we perform a two-sample Kolmogorov-Smirnov test on the feature data to determine for each feature and each malware sample whether the feature exhibits a change in distribution after the system is infected [39]. Here, we remove from consideration all features that do not exhibit a change in distribution for at least one of the malware samples.

For the second stage, we eliminate those features whose data are not useful for malware detection using Page's CUSUM test even though they exhibit a change in distribution. As a simple example, one of the sensors reports the uptime of the VM, which is always higher after the malware executes and thus identified as one of the sensors whose distribution is different after infection. We remove from consideration all of the features that exhibited such behavior, including features that exhibited monotone or near-monotone increasing behavior.

In the final stage, we use the local detectors to perform Page's CUSUM as described in Section IV on the remaining features. We perform the test as previously described: first training the detector when the guest OS is first started and then performing detection on the remaining data. Sensors exhibiting a high false alarm rate, *i.e.,* sensors that frequently indicate the host is infected when it is clean, are removed from consideration.

After feature reduction, 339 features remain that are used for detection. The remaining features include mainly processor and network performance indicators. On the processor side, the remaining features include:

- Processor hypercalls/sec,
- Interrupt hypercalls/sec,
- Large page TLB fills/sec,
- Percent privileged time,
- MSR accesses cost, and
- CPUID instructions cost.

And on the networking side, the remaining features include:

- Outbound connections/sec,
- Miniport Send Cycles/sec,
- Stack Receive Indication Cycles/sec, and
- NDIS Receive Indication Cycles/sec.

C. Detection performance

In this section, we examine the detection performance of the MDS in terms of overall detection accuracy and detection delay. The detection results presented in this section are obtained using 375 iterations of the test sequence described in Figure 5. Since there are only 200 malware samples used for testing, this means that multiple malware samples are

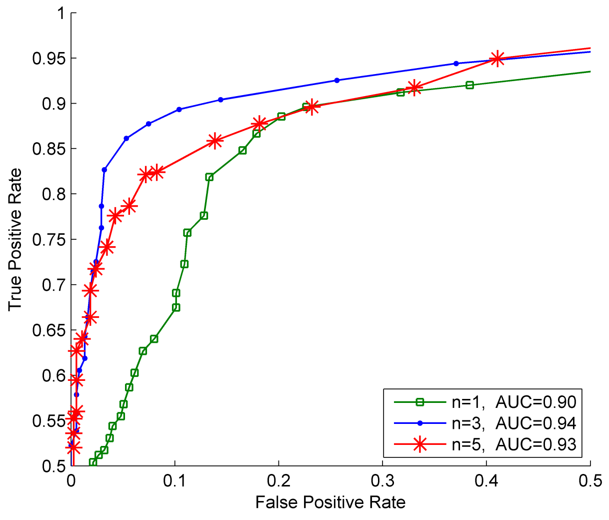

Fig. 6. Receiver Operating Characteristic of the MDS

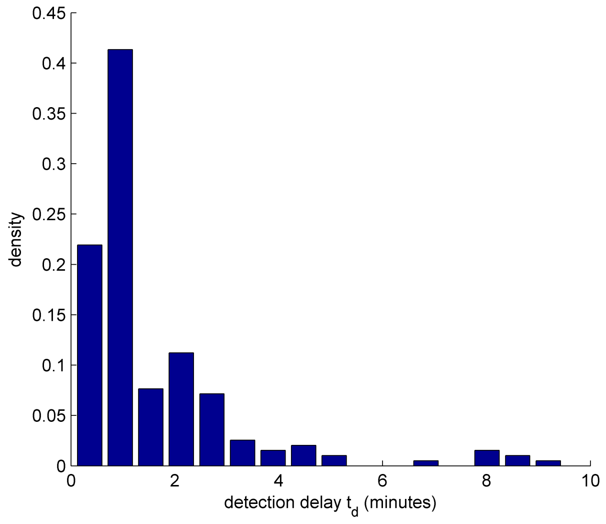

Fig. 7. Distribution of detection lags

used twice, each time with a different start time and different ordering of the background load generated by the Drupal test suite.

The testing results presented in this section are obtained using the following parameters: The parameter estimates are determined during a 7 minute training period separated into 20 second blocks. We configure the standard deviation multiplier for the change magnitude to be $n = 3$, setting $\delta_\mu = 3\sigma_{\mu_{\text{ML}}}$ for the mean and $\delta_{\sigma^2} = 3\sigma_{\sigma^2_{\text{ML}}}$ for the variance. Finally, the number of standard deviations above the mean to set the detection thresholds $h^{(a)}$ was determined experimentally by determining the smallest number of standard deviations that would result in no false alarms in data sets used for feature reduction. The number of standard deviations is uniquely determined for each sensor. Finally, a 5 minute reset delay is used at the DFC after which any local detector indicating a positive detection is reset if no global detection has been made.

We evaluate the overall accuracy of the MDS in terms of its receiver operating characteristic (ROC), a plot of the true positive rate vs. the false positive rate of the system. The curve is generated by varying the threshold k of the DFC. Here, we define the true positive rate as the fraction of the data sets that the MDS correctly identifies as infected after the malware executes. Conversely, we define the false positive rate as the fraction of the data sets that the MDS incorrectly identifies as infected when the host is in a clean state.

The ROC is presented in Figure 6 and shows only the detail of the upper-left quadrant of the ROC. For comparison, we show three ROC curves for different change magnitudes: $n = 1$, $n = 3$ and $n = 5$. The total area under the curve (AUC) of the ROC is a measurement of the overall accuracy of the detector, where a perfect detector has an AUC of 1. The AUC of the MDS when $n = 3$ is the highest, at 0.94. We are also interested in determining the highest detection rate we can achieve without experiencing any false alarms: 0.52. This number indicates that the MDS was able to correctly identify that the VM was infected for 52% of the data sets without any occurrence of false alarms, using a DFC threshold of $k = 21$.

We continue our analysis of the performance of the MDS by examining the detection delay t_d. For this analysis, we use the DFC detection threshold $k = 21$ that gives rise to the 52% detection rate described above and measure the detection delays as the elapsed time between when the malware is executed and when the MDS correctly identifies the host as infected. The median detection delay t_d is 96 seconds and 90% of the malware are detected in under 3 minutes.

A histogram of the detection delays measured in minutes for each of the data sets is shown in Figure 7. The histogram indicates that for some data sets the MDS took up to 10 minutes to detect the offending malware. These data sets likely involved malware samples that were less aggressive at their onset. For example, one such malware sample was observed waiting several minutes before spawning new processes and performing malicious tasks, presumably to avoid being noticed by an administrator monitoring the process activity on the system. This finding points to the possibility that there may have been other malware samples that the MDS would have detected if the host continued to run longer than 10 minutes after infection, leading to an increase in the overall detection accuracy of the system.

D. Comparison to related work

Lanzi *et. al* [15] provided what is probably the most comprehensive evaluation of the state of the art in behavioral malware detection techniques to date, evaluating the accuracy of two different detection techniques on data captured from multiple production computer hosts. They provide detection results for both an n-gram system call sequence detector [12] and their own malware detector that uses abstracted models of filesystem and registry usage, called access activity models. Since Lanzi *et. al* did not characterize the overall accuracy of their detector in terms of its ROC, we compare our results using the detection rates achieved at a zero false alarm rate. At zero false alarms, our MDS with a 52% detection rate outperforms the n-gram detector with a detection rate under 40%.

However, our MDS underperformed the access activity model detector, which exhibited a detection rate of 90%. Reasons for this performance discrepancy include differences in the time period over which training data was captured – more than 40 hours for the access models compared to 7 minutes for our MDS – and the manual adjustments performed to the access activity models after they were learned from the training data to mitigate the occurrence of false alarms. Furthermore, we believe our MDS, whose sensors span a wide array of system functionality, including CPU, memory, storage, and service usage patterns to be complementary to the the access activity models, which are designed to detect malware that violate file and registry access policies. Other comparable work includes work by Moskovitch *et. al* [21] which aims to detect network worms by using a similar set of sensors and various machine learning algorithms. However, the limited malware corpus considered by Moskovitch *et. al* (5 network worms) precludes meaningful comparison.

E. Discussions

The case study was designed to determine whether the described MDS architecture coupled with the selected features is effective at detecting the execution of state-of-the-art malware on a modern operating system under heavy and heterogeneous computational load. Three design choices – the decentralized detection architecture, the use of Page's CUSUM test, and the fusion rule used at the DFC – warrant further discussion.

The decentralized detection architecture was chosen under the assumption that the function and implementation of malware determine the subset of features that are perturbed when the malware executes. In the case study, we achieved a 52% detection rate with 0 false alarms when only 21 parameters from the 339 features exhibited a detectable change. Furthermore, the subsets of features exhibiting a change for each malware sample varied, with all 339 features being perturbed by at least one malware sample. Additionally, the ROC curve shows that increasing the detection threshold leads to a decrease in detection rate, indicating that the number of sensors perturbed by the malware also differs from sample to sample.

Page's CUSUM test was chosen because it is among the computationally least complex algorithms for change-point detection that can handle a problem where only the parameters before the change are known. However, it requires that we assume a parametric distribution for the feature data and limits us to detecting only changes in parameter for the chosen distribution. Statistical goodness of fit tests were used to determine that the majority of the feature data could be accurately approximated by a normal distribution; however, there were features that were not well-described by a normal distribution. The application of non-parametric detection techniques at the local detector level could possibly overcome this limitation and provide more accurate detection results [43].

The k out of N fusion rule is used to establish a baseline of the performance that can be achieved using the described system. We expect that the use of fusion rules that consider the statistics of the local detectors and the sequential nature of the data will lead to improved overall detection performance [40], [41], [42]. The DFC may also be extended to perform classification to help guide mitigation and post-mortem analysis. For example, since we know that the subset of perturbed features differs for each malware sample, we may be able to map from the perturbed features to an existing malware taxonomy.

The features chosen for this study were chosen because they monitor a wide variety of features of the underlying operating system, and the detection results indicate that a subset of 339 of them provided adequate information to infer the execution of malware on a live computer host. We expect that in order to make the system more accurate and applicable to a wider variety of computer hosts, the feature set should be expanded to other layers, including sensors monitoring individual applications and services, network usage, and the host OS running the hypervisor. Additionally, it may prove useful to extract features from system call, filesystem, registry, or network traces, as such traces have been shown to be useful data sources for malware detection [12], [15].

Finally, we briefly discuss our choices for the amount of training time, the block size used during training, the detection thresholds $h^{(a)}$, and the standard deviation multiplier n used to determine the change magnitudes. These values were arrived at experimentally, by considering a variety of choices for each parameter and comparing the AUC of the associated detection results. Changing each parameter from the stated values generally resulted in a marginal decrease in overall detection accuracy, with the exception of the training time. For example, Figure 6 shows that changing the standard deviation multiplier from $n = 3$ results in an overall decrease in detection accuracy. Increasing the training time leads to an increased detection accuracy, although the percent improvement quickly declines as the training time increases. For example, increasing the amount of training from 7 to 8 minutes led to an increase in the AUC of less than 1%.

VI. Summary and Conclusions

This paper presents a novel application of Page's CUSUM test to the malware detection problem, using the test as part of a decentralized malware detection system (MDS) that uses a data fusion center (DFC) to process decisions made by the local detectors to determine whether a host is infected with malware. The paper poses the malware detection problem as a quickest detection problem and describes a novel application of sequential detection techniques and data fusion rules to infer the presence of malware.

The paper presents the results of a case study designed to test the effectiveness of the described MDS in detecting malware on a virtual machine host experiencing heavy and diverse computational load. The results demonstrated that the MDS was capable of quickly detecting a majority of the malware with no false alarms on the described testbed. We believe these results demonstrate the promise of both quickest detection algorithms and the described MDS architecture as potential solutions to the malware detection problem.

We identified potential avenues for future work, including an exploration of alternate fusion rules, expansion of the system to perform classification, and the inclusion of additional features to use for malware detection. Furthermore, additional testing of the MDS under different load conditions and on different hosts would be useful for determining the applicability of the described approach to other systems. We will continue to explore some of the questions posed here in our continuing effort to build an adaptive host-based malware detection system that is robust in detecting new malware that are behaviorally similar to known malware and obfuscated variants of known malware.

References

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

[2] B. Pollak, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.

[3] O. Sukwong, H. Kim, and J. Hoe, "An empirical study of commercial antivirus software effectiveness," *Computer*, vol. PP, no. 99, p. 1, 2010.

[4] M. Christodorescu and S. Jha, "Testing malware detectors," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 34–44, July 2004.

[5] C. Nachenberg, "Computer virus-antivirus coevolution," *Commun. ACM*, vol. 40, no. 1, pp. 46–51, 1997.

[6] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, vol. 4, pp. 251–266, 2008, 10.1007/s11416-008-0086-0.

[7] D. Gryaznov and J. Telafici, "What a waste - the av community dos-ing itself," in *Proc. of the Virus Bulletin Conf.* McAfee Avert Labs, 2007.

[8] M. Basseville and I. V. Nikiforov, *Detection of Abrupt Changes - Theory and Application*. Prentice-Hall, 1993.

[9] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, "Data mining methods for detection of new malicious executables," in *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, 2001, pp. 38 –49.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 32–46, 2005.

[11] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, dec. 2007, pp. 421 –430.

[12] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[13] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, 2011.

[14] T. Lee and J. J. Mody, "Behavioral classification," in *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[15] 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*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 399–412.

[16] 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*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 116–127.

[17] 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: USENIX Association, 2006.

[18] 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*. Berkeley, CA, USA: USENIX Association, 2007, pp. 18:1–18:14.

[19] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, may 2007, pp. 231 –245.

[20] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 61–74.

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

[22] 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*, pp. 1–30, 2011, 10.1007/s10844-010-0148-x.

[23] 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*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 109–118.

[24] A. Shabtai, E. Menahem, and Y. Elovici, "F-sign: Automatic, function-based signature generation for malware," vol. PP, no. 99, 2010, pp. 1 –15.

[25] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, p. 48, 2007.

[26] A. G. Tartakovsky, B. L. Rozovskii, R. B. Blaek, and H. Kim, "Detection of intrusions in information systems by sequential change-point methods," *Statistical Methodology*, vol. 3, no. 3, pp. 252 – 293, 2006.

[27] H. Wang, D. Zhang, and K. Shin, "Change-point monitoring for the detection of dos attacks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 4, pp. 193 –208, oct.-dec. 2004.

[28] Xin and Xu, "Sequential anomaly detection based on temporal-difference learning: Principles, models and case studies," *Applied Soft Computing*, vol. 10, no. 3, pp. 859 – 867, 2010.

[29] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. pp. 100–115, 1954.

[30] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 186–205, August 2000.

[31] H. V. Poor and O. Hadjiliadis, *Quickest Detection*. Cambridge, 2009.

[32] J. Chen and A. K. Gupta, *Parametric Statistical Change Point Analysis*. Birkhauser, 2000.

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

[34] D. Teneketzis, "The decentralized quickest detection problem," in *Decision and Control, 1982 21st IEEE Conference on*, vol. 21, dec. 1982, pp. 673 –679.

[35] Dionaea. [Online]. Available: http://dionaea.carnivore.it

[36] Microsoft, "Microsoft server products." [Online]. Available: www.microsoft.com

[37] "Drupal content management system." [Online]. Available: http://drupal.org/

[38] B. Quintero. (2012) Virustotal. [Online]. Available: www.virustotal.com

[39] N. Smirnov, "Tables for estimating the goodness of fit of empirical distributions," *Annals of Mathematical Statistics*, 1948.

[40] R. Blum, S. Kassam, and H. Poor, "Distributed detection with multiple sensors ii. advanced topics," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 64 –79, jan 1997.

[41] W. Chang and M. Kam, "Asynchronous distributed detection," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 818 –826, jul 1994.

[42] A. Hussain, "Multisensor distributed sequential detection," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 698 –708, jul 1994.

[43] J. Thomas, "Nonparametric detection," *Proceedings of the IEEE*, vol. 58, no. 5, pp. 623 – 631, may 1970.