Learning Process Behavioral Baselines for Anomaly Detection

(Regular Paper)

Ahmed M. Fawaz and William H. Sanders
Coordinated Science Laboratory (CSL)
University of Illinois at Urbana-Champaign
{afawaz2, whs}@illinois.edu

***Abstract*—Intrusion resilience is a protection strategy aimed at building systems that can continue to provide service during attacks. One approach to intrusion resilience is to continuously monitor a system's state and change its configuration to maintain service even while attacks are occurring. Intrusion detection, through both anomaly detection (for unknown attacks) and signature detection (for known attacks) is thus a crucial part of that resilience strategy. In this paper, we introduce KOBRA, an online anomaly detection engine that learns behavioral baselines for applications. KOBRA is implemented as a set of cooperative kernel modules that collects time-stamped process events. The process events are converted to a discrete-time signal in the polar space. We learn local patterns that occur in the data and then learn the normal co-occurrence relationships between the patterns. The patterns and the co-occurrence relations model the normal behavioral baseline of an application. We compute an anomaly score for tested traces and compare it against a threshold for anomaly detection. We evaluate the baseline by experimenting with its ability to discriminate between different processes and detect malicious behavior.**

***Index Terms*—anomaly detection, behavioral baseline, intrusion detection system, intrusion resilience, kernel monitoring**

I. INTRODUCTION

As today's computers are involved in every aspect of our lives, they are attractive targets for attacks. These attacks are no longer limited to cyber assets, but also extend to the systems they control, which raises the need for proper and secure protection mechanisms. Hardware- and software-based mechanisms such as access control [1], sandboxing, and protection rings [2] have been deployed to address security threats. Even with such protections in place, compromises are still inevitable, because vulnerable software offers entry points for penetration. In effect, attackers and defenders are locked in an arms race; for example, modern firewalls, intrusion detection systems (IDSes), and anti-virus software must always be kept up to date to face ever-evolving malware [3], [4].

In that arms race, it is prudent to build several layers of protection to make systems more resilient to intrusions. Intrusion resilience [5], inspired by fault tolerance, aims to build systems that can maintain their mission despite compromises. An intrusion-resilient protection mechanism is one that employs a strategy of continuous monitoring and response. The protection mechanism reacts to changes in the security state by reconfiguring the system while maintaining an acceptable service level.

Intrusion detection, through either anomaly detection (for unknown attacks) or signature detection (for known attacks), is often deployed as the monitoring component of resilience strategies. Unfortunately, intrusion detection systems are notoriously noisy (with a high rate of false positives), which overwhelms both operators and decision algorithms [6], making them the Achilles heel of resilience strategies.

Still, anomaly detection is the most effective strategy against unknown attacks. State-of-the-art black-box anomaly detection systems (see Section VII) in modern OSes rely on execution traces of running processes [7] as an alphabet to detect anomalous subsequences. Anomalies are detected when events co-occur in a manner different from normal. This approach is particularly effective in detecting arbitrary code execution attacks, unauthorized behavior, and other policy violations that change a running process without modifying binaries; such attacks defeat signature-based mechanisms.

However, most behavior traces proposed in the literature use an alphabet that is hard to collect in most modern operating systems (such as system calls, and function calls). Since intrusion detection systems run as independent processes in OSes, their developers resort to modifying the kernel by overwriting the addresses of functions in order to intercept the events from other running processes. Ironically, attackers utilize the same techniques in developing rootkits [8]. As a result, modern operating systems include several mechanisms to prevent tampering with kernel data structures (such as Kernel Patch Protection in Windows) [9]. Those protections make it impossible for intrusion detection systems to collect system calls without tampering with the security of the system being protected; this creates a gap in host-based security protection. Therefore, there is a need for host-based intrusion detection systems that are practical and safe to deploy. In this paper, we address the following related question: *can we utilize available information in the operating system, without modifying its internals to achieve accurate anomaly detection?*

During an attack, an application's behavior deviates from its normal behavior, which can be modeled as a baseline. There are two challenges in modeling behavioral baselines: (1) deciding on the data sources to monitor, and (2) extracting

978-1-5090-5652-1/17/$31.00 ©2017 IEEE

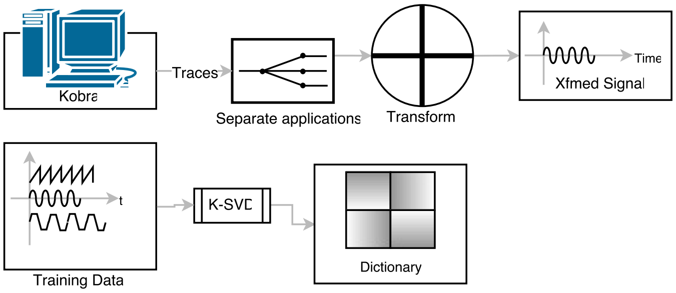


Fig. 1. High-level description of our approach.

features from said data. First, models extracted from low-level data (e.g., network usage patterns) might not be discriminating, while models with high-level data (e.g., system calls and function calls) are costly to build and maintain and might not be accurate because of their high dimensionality. Second, feature extraction involves projection of the collected data onto a vector space basis. The selected features should reveal the subtleties in behavior that enable anomaly detection.

Figure 1 shows KOBRA, our practical kernel semi-supervised anomaly detection system that we developed for Microsoft's Windows operating systems (Windows 7+). To address the first challenge, we focus solely on the low-level information exposed by the kernal via public filters and APIs, without requiring instrumenting or "hacking" of the kernel. We observe that much of this information, such as network and file system usage patterns, evolves over time, and thereby provides a high-fidelity feature set that may be used to uniquely identify running applications and variations thereof. In particular, we found that we can build accurate behavioral baselines to detect anomalous process behavior due to arbitrary code execution attacks.

We collect information from the kernel using KOBRA. KOBRA filters the data by application to create per-application event traces. We transform each trace into a complex-valued discrete-time signal by mapping discrete events onto the z-plane, as explained in Section III-A.

Instead of explicitly specifying the vector set of basis vectors(features), we address the second challenge by learning the set of basis vectors using a training set of normal behavioral traces. The training set is constructed using overlapping subsequences that are obtained by sliding a window over the transformed time signal. We use a sparse dictionary-learning algorithm to learn local patterns in the data. Then we learn a second set of basis vectors using latent semantic analysis; these vectors encode co-occurrence relations between local patterns in the training data. We compute an anomaly score as the reconstruction error when approximating the data using the learned set of basis vectors. Normal behavior will be accurately represented when projected onto the learned set of basis vectors; anomalies, which occur in different patterns from normal behavior, cannot be accurately represented and thus will lead to high anomaly score.

In this paper, we evaluate the effectiveness of the learned behavioral baselines for anomaly detection. We consider process-execution-hijacking attacks over a wide range of applications, such as the VLC player. Process hijacking is a technique used by malware to perform malicious tasks without having persistent processes that can easily be detected. Our study shows that the learned baselines for different applications did not share similar local patterns, confirming that they are suitable for modeling application behavior. Moreover, we evaluated the effectiveness against attacks by weaving attack data into normal behavioral traces. We considered two types of shellcode attacks and the detection accuracy was around 95% with a low false positive rate. Finally, we evaluated the performance of KOBRA for data collection and online anomaly detection; in general we observed low overhead during the data collection phase, and the system was stable for online anomaly detection.

In summary, the contributions of this paper are as follows:

- KOBRA, a Windows kernel-monitoring engine and an online anomaly engine that collects events correlated with running processes;
- A novel transformation from discrete event traces to a complex-valued discrete-time signal, and
- A method utilizing sparse representation and latent semantic analysis to baseline application behavior and detect anomalous behavior.

The remainder of this paper is organized as follows. Section II highlights the kernel events that are collected by KOBRA; Section III introduces our transformation from KOBRA traces to time signals and dictionary learning; Section IV evaluates the accuracy of our baselines; Section V describes the implementation of KOBRA; Section VI highlights the assumptions and limitations of our approach; Section VII describes related work in anomaly detection and kernel monitoring; and Section VIII provides concluding remarks and plans for future work.

II. DATA COLLECTION

We developed KOBRA as a kernel-monitoring engine with realistic monitoring capabilities for modern operating systems. The engine collects process behavior from the kernel without any modifications to the kernel or the processes. The process behavior includes network operations, file operations, pipe communication, and process creation events. Processes are labeled with unique IDs. Events from each process are tagged with a timestamp and stored as a data stream.

A data stream is a sequence of events that represents the observed process behavior over time. Formally, an event is a pair of an ID and a time of occurrence, $e_t = (E, t)$, where $t \in \mathbb{N}$ and $ID \in \mathcal{E}$, the universe of all process events. The event sequence is a totally ordered set of events $(E, <)$, where E is the set of events $E = (e_1, e_2, e_3, \ldots)$, and $e_i < e_j$ if $e_i.t < e_j.t$. Table I shows the types of events we are currently collecting. A process creation event is a fork event for which KID is a unique process identifier assigned by KOBRA. Network events are either connect events or send and receive events for which IP is the remote address. The

TABLE I
List of possible events in a host.

Type	Event	Event
Process	Create	`{KID} {devos:proc} {name}`
Network	Connect	`{IP} {net:remote}`
	Send	`{KID-IP} {net:send} {size}`
	Receive	`{KID-IP} {net:recv} {size}`
Storage	New	`{FID} {devos:file}`
	Read	`{KID-FID} {devos:read} {size}`
	Write	`{KID-FID} {devos:write} {size}`

send/receive events are tagged with the buffer size. Storage events are those that manipulate files. They include opening of a new file, and read and write events. The read/write events are tagged by the buffer size and by the FID, a unique file identifier assigned by KOBRA. Each event is tagged by the process that caused it. KOBRA outputs a data stream for analysis (training and behavioral baselining) and data logging. Section IV-C evaluates the overhead caused by the data collection. Section V contains implementation details of KOBRA.

In this section, we described the events collected by KOBRA. In the next section, we specify how we process the data stream and convert it to a complex-valued discrete-time signal. The signal is used to learn the behavioral baselines of applications for anomaly detection.

III. Data Analysis

KOBRA generates a stream of data due to process behavior. We use the normal behavior data to learn a baseline for anomaly detection. We start by transforming the data stream into a complex-valued discrete-time signal. Specifically, each event is transformed into a complex number, and the sequence of time-stamped complex numbers generates the discrete-time signal. Then we construct a training set from collected normal behavior traces. We use the training set to learn a behavioral baseline by constructing a sparse representation dictionary. Then we use sparse representations of the training set to construct latent semantic analysis (LSA) matrices. Both the dictionary and LSA matrices represent a process's normal behavior. Finally, the anomaly detection algorithm computes an anomaly score using the sparse reconstruction error and LSA reconstruction error. Detection thresholds are assigned as the 90th percentile of the anomaly scores of the normal data.

A. Data Stream to Complex-Signal Transformation

We transform KOBRA's data stream into a time signal for analysis. First, we divide the stream into per-process/per-application traces using a process tag in each event. Next, we convert each event in a trace to a complex-valued number. Finally, we combine the complex values to form a discrete-time signal.

We start by dividing the data stream by application to generate per-application traces, Figure 2 shows part of a VLC trace. Events are tagged by the unique KOBRA Identifier (KID) and application ID to identify the running process and the application, respectively. (Each application might have more than one running process.)

The event-to-complex-value transformation was inspired by constellation diagrams in digital modulation schemes. The basic idea is to map discrete events to the complex space, $f : e \rightarrow x, e \in \mathcal{E}, x \in \mathbb{C}$. Equations 1 and 2 compute the phase and magnitude of the transformed event respectively. The complex plane is divided into N equal angular zones, where N is the number of types of events (in our example, $N = 4$); each type of event is mapped to a zone. In equation 1, z is a function that maps an event type to the appropriate zone $z : e \rightarrow k, e \in \mathcal{E}, k \in [0, N-1]$ as defined in Figure 3b. e.Obj.ID is a counter assigned to each unique instance of an object (file or IP). The phase of each event within each zone is assigned according to the total number of unique instances. In equation 2, the magnitude of each number (e.Obj.size) is the normalized size of the magnitude of the event (number of bytes transferred) per region.

$$\angle f(e) = \left(\frac{e.Obj.ID}{\max_{x \in \mathcal{E}} x.Obj.ID} + z(e.ID)\right) \times \frac{2\pi}{N} \qquad (1)$$

$$|f(e)| = \frac{e.Obj.size}{\max_{x \in \mathcal{E}} x.Obj.size} \qquad (2)$$

We transform the trace into a complex-valued signal by transforming each event, e, into a complex value such that $\theta = \angle f(e)$ and $r = |f(e)|$, and then assigning the value to the appropriate position in the complex-valued signal ($\mathtt{tr}$) $\mathtt{tr}[e.t] = r.exp(j.\theta)$.

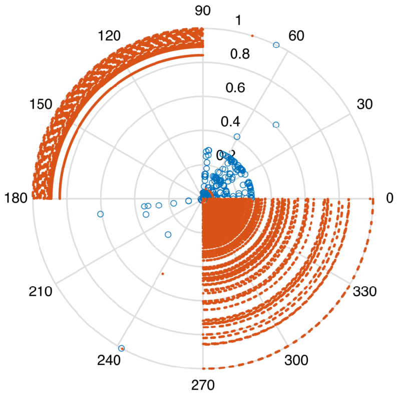
The zone mapping allows us to fuse heterogeneous sources of data into one signal. Figure 3 shows an example of the transformation; the plot shows the complex-time signal with the time domain collapsed. Each data point is plotted on the z-plane. We compare the events extracted from `explorer.exe` (blue ○) and the Apache server (orange ●). The Apache server's behavior has high activity in the network zones (more biased to the send zone); in this instance WordPress is using MySQL for data storage. On the other hand, `explorer.exe` has more activity in the file read zone (as it is the file browser).

Traditionally, anomaly detection systems encode events as a sequence of integers. Those encodings remove essential timing information and other semantics that are important for behavioral analysis, while our complex time signal preserves them. The semantics and timing reflect the behavior of the application (functionality) and implementation details (buffer sizes, sleep intervals, etc.). By keeping the semantics, we protect our anomaly detection method against mimicry attacks that only change function call parameters [10]. Moreover, even though the transformation is lossy, it preserves important frequency information that helps in baselining processes' behavior. Our anomaly detection algorithms will exploit the information in the signals to learn the behavioral baselines.

Given a set of normal behavior traces of an application $Y = \{tr[k]\}_i$, we construct the training set by applying a sliding window over the time signals. By using overlapping

```
...
870 {"VID.mp4" :?: 2044}{devos:read} {512}
895 {"VID.mp4" :?: 2044}{devos:read} {512}
923 {"VID.mp4" :?: 2044}{devos:read} {4096}
...
```

Fig. 2. Data stream output while VLC is playing a video. The output shows VLC reading the video file over time. The first column is the timestamp; the second column is the filename and the process ID; the third column is the type of operation (a file read); and the last column is the buffer size. 2044 is the ID of the process running the player. The file path and timestamp are truncated to fit in the column.



(a)

z	Event	Range
0	Read File	$[0\ ,\ \frac{\pi}{2}]$
1	Write File	$[\pi\ ,\ \frac{3\pi}{2}]$
2	Net. Recv.	$[\frac{\pi}{2}\ ,\ \pi]$
3	Net. Send	$[\frac{3\pi}{2}, 2\pi]$

(b)

Fig. 3. (a) Time-collapsed representation of execution of `explorer.exe` (blue ○) and Apache server (orange ●). (b) The angular zones used for this representation.

subsequences, we alleviate the issue of time shifts in the signal. Specifically, the training set is constructed by running a sliding window on each trace $tr_i[.]$ to obtain overlapping subsequences,

$$\mathcal{TS}^{(i)} = \left(ts_1^{(i)}, \dots, ts_n^{(i)}\right),$$

where $ts_k^{(i)}$ is a subset of the trace $tr_i[.]$ that spans n points (the window size) $ts_k^{(i)} = (tr_i[k], \dots, tr_i[k+n])$. The training set is a concatenated set of overlapping subsequences arranged in a matrix with n columns. Figure 4 shows a HeatMap of the training set from Chromium and VLC; it is easy to see the difference between the patterns emerging from the two datasets. Those patterns will be learned by the behavioral baselines.

B. Learning Sparse Representation

The output of the transformation is a time signal that we want to sparsely represent to detect local patterns. Sparse approximation assumes that an input signal $y \in \mathbb{R}^n$ can be described in terms of an overcomplete linear system.

$$y \approx \mathbf{D}x, \tag{3}$$

where $\mathbf{D} \in \mathbb{R}^{n \times p}$ $(n \ll p)$ is called the *dictionary* and $x \in \mathbb{R}^p$ is the *sparse approximation*. The recovery of a sparse approximation problem is represented as an optimization problem.

$$x^* = \arg\min_x \|y - \mathbf{D}x\|_2^2 \ \ s.t.\ \ \|x\|_0 \leq T \tag{4}$$

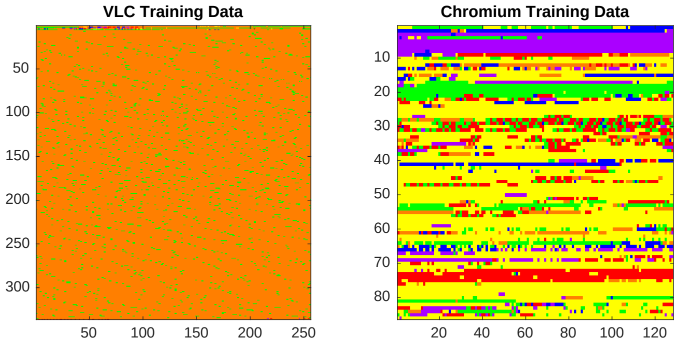


Fig. 4. The HeatMaps of the training sets for VLC and Chromium.

In this optimization, we want to find the representation that minimizes the approximation error $\|y - \mathbf{D}x\|_2$;The minimization is subject to the number of nonzero elements in the approximation T, referred to as *sparsity*. The pseudonorm ℓ_0, $\|\cdot\|_0$, counts the number of zero elements in the vector. We typically want $T \ll p$. Equation 4 is a combinatorial optimization problem; researchers have proposed sub-optimal greedy algorithms to solve the problem. In this paper, we use the Orthogonal Matching Pursuit (OMP) algorithm [11].The atoms of a dictionary, are set of basis vectors of a space, are the local patterns. The sparse representation of a signal is the decomposition of this space onto those atoms. Thus, the dictionary choice affects the resulting sparse representations. In this work, we are interested in choosing a dictionary with atoms specifically designed for normal behavior traces. That is, we seek atoms that represent unique local patterns that exist in the data. For that purpose, we learn the dictionary using K-SVD [12]. K-SVD is an algorithm that iteratively learns a dictionary by solving the problem in equation 5.

$$\mathbf{D}^* = \arg\min_{\mathbf{D}} \sum_{i=1}^{N} \min \left\{ \|\mathbf{D}x_i - y_i\|^2 + \lambda \|x_i\|_1 \right\} \tag{5}$$

The set of training data has to be large enough for K-SVD to learn normal local patterns. The learned dictionary $\mathbf{D}^*$ is used for testing new data for anomalies. The sparse representation of the training data, $X_{\mathcal{TR}}$, is used for semantic analysis to learn the normal co-occurrence relationships using latent semantic analysis.

C. Learning Co-occurrence Relationships

The next step is to learn co-occurrence relations between local patterns in the signal. We use latent semantic analysis (LSA), a dimensionality reduction method used in NLP. In LSA for NPL, a term matrix is decomposed and approximated. The term matrix is matrix that counts the frequency of words (from a corpus) in documents to be studied. The sparse representation of a vector similarly assigns frequencies of local patterns in the subset to be studied. In that sense, words in the corpus and local patterns are equivalent. The sparse representation vectors of the training data are arranged in a matrix X^*. LSA is performed using the following steps:

1) Factorize the sparse representation matrix of the training data using Singular Value Decomposition (SVD) $X^* = U\Sigma V^T$.
2) Approximate the matrix decomposition by keeping the eigenvectors of the k-largest eigenvalues in Σ such that $X^* = U_k \Sigma_k V_k^T$.
3) Transform to a lower dimension $\hat{x} = \Sigma_k^{-1} U_k^T x$.
4) Reconstruct the sparse representation $\tilde{x} = U_k \Sigma_k \hat{x}$.

The decomposition matrices, U_k and Σ_k, are the co-occurrence baseline for the application. The anomaly score is computed as $(x - \tilde{x})^2$, which is the reconstruction error due to the latent semantic analysis. If the behavioral traces are anomalous (not part of the normal trace), the relationships within the traces cannot be represented, and thus will have a high reconstruction error.

D. Behavioral Baseline for Anomaly Detection

We learned the per-application normal behavior model using sparse representation and latent semantic analysis $< D^*, \Sigma_k, U_k >$. Our behavioral model learns two modes: (1) local patterns and (2) co-occurrence of the local patterns in the normal behavior trace. The local patterns are extracted by learning a sparse representation dictionary using K-SVD. The sparse representations of the reference signal are used to learn a co-occurrence model of the normal behavior relative to the local patterns using LSA. The detector will use both modes in the behavior model for detection of anomalies. The detector uses the modes in two stages to detect anomalies. In the first stage, the sparse representation of input signal y is constructed using the learned dictionary $\hat{x} = \arg\min_x |D^* x - y|$; if the sparse reconstruction error (SRE) $|y - D^* x^*|^2$ is higher than a threshold λ_{SRE} then an alert is issued. In that case, the normal local patterns could not effectively represent the behavior being tested, and thus, the behavior is marked as an anomaly. However, if the SRE is below the threshold then the latent semantic representation of the sparse representation vector x^* is computed, $\tilde{x^*} = U_k \Sigma_k x^*$. The latent reconstruction error (LSE) is computed as $|x^* - \tilde{x^*}|^2$. An alert is issued if the LSE is greater than a threshold λ_{LSE}. The two-stage process is used to avoid expensive computations; if the SRE is high, then we do not need to compute the LSE. The thresholds are selected as a function of the SRE and the LSE of the training data. Algorithm 1 lists the procedure for anomaly detection using behavioral baseline $< D^*, \Sigma_k, U_k >$ for any application trace x.

This section described the method used to transform a graph stream into complex-valued discrete-time signal. It introduced the behavioral baseline as a learned dictionary for sparse representation and the anomaly detection method. In the next section, we evaluate the effectiveness of the behavioral baselines in detecting anomalies.

IV. Evaluation

In this section, we describe our strategy for evaluating anomaly detection using the behavioral baselines. We started by collecting data from different applications; then we wove attack traces into normal behavior and computed the detection rates of the anomaly detection method. Our objectives were to verify the following: (**O1**) that the learned local patterns and co-occurrence relations are unique to each application, and (**O2**) that use of the baselines is capable of detecting anomalous behavior including malicious behavior. We devised multiple experiments to evaluate the baselines.

Algorithm 1 Anomaly detection procedure using LSE

```
1) Given $x$ a subsequence of a behavior trace for application $A_1$ with baseline $(\mathbf{D_{A_1}}, \Sigma_k, U_k)$
2) Compute the sparse representation using OMP as $y^* = \arg\min_y \|\mathbf{D_{A_1}} y - x\|_2 st \|y\|_0 \leq T$
3) Compute the sparse reconstruction error (SRE) as $\delta_{SRE} = \|\mathbf{D_{A_1}} y^* - x\|_2$
4) Compute the latent semantic representation $\hat{x} = \Sigma_k^{-1} U_k^T$
5) Compute the latent semantic error (LSE) $\delta_{LSE} = \|U_k \Sigma_k \hat{x} - x\|_2$
6) Check $\delta_{SRE} \geq \lambda_{SRE}$ and $\delta_{LSE} \geq \lambda_{LSE}$
```

In Experiment set 1, we collected normal behavioral traces from different applications, and then learned the baselines for all the applications. We compared the similarities between baselines and the level of effectiveness in discriminating between the applications. In Experiment set 2, we evaluated the ability of the baselines to detect malicious behavior. We wove attack behavior into the behavior trace of one application.

In the following we explain our experiments and the results. Finally, we study the performance overhead of Kobra.

A. Experiment 1: Comparing Applications and Baselines

In the first set of experiments, we wanted to verify that the learned behavioral baselines are unique for each application. For this purpose, we collected behavioral information for multiple applications using Kobra. Then we learned the behavioral baselines of the application, and finally we tested the effectiveness of the anomaly score in discriminating between applications. We selected the following applications for training:

VLC: We obtained 20 VLC (version 2.2.1) execution traces by playing local videos of various lengths and formats.

Web server: We set up Apache (version 2.4.9) with php and MySQL (version 5.6.17). The Web server has a set of files for download and a web blog application (WordPress). We performed a stress test on the Web server by sending it random requests with a varying rate. The timing distribution followed a Poison process with rate $\lambda = 20$. The requested content was drawn from a uniform distribution over the index of all accessible data. We ran the tests for 4 hours.

OS Processes: We scraped the traces generated by Kobra in the VLC and Web server setups for behavior traces generated by running Windows processes, including `svchost.exe` and `explorer.exe`. Most applications run in different modes; for example, VLC can be used to stream video or play local files. In this work,

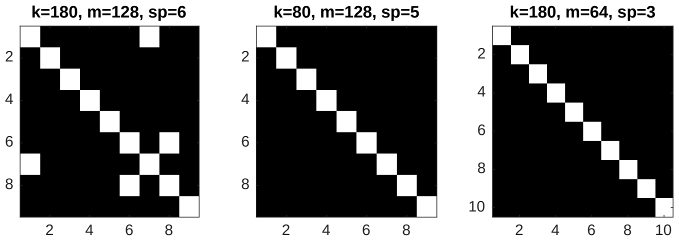


Fig. 5. Similarity between behavioral baselines for different parameters.

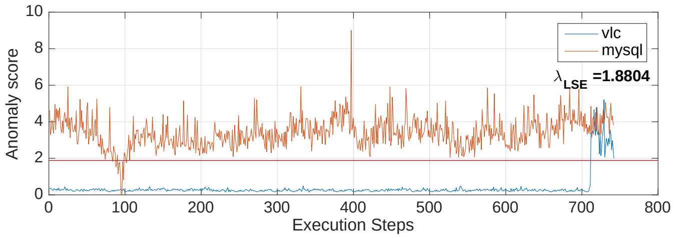


Fig. 6. The SRE anomaly score for `mysql.exe` compared to `vlc.exe` using the `vlc.exe` behavioral baseline.

we learned a behavioral baseline per mode of operation. The mode of operation is to be detected by finding the baseline that has the lowest anomaly score.

For each application, we assembled the training set in a matrix with a sliding window of size $n = 32$. We learned the behavioral baseline of each application using the method we highlighted. We set the number of local patterns to $m = 180$, the sparsity to $T = 5$, and the LSA approximation to $k = 30$.

1) Comparing Baselines: A good discriminating baseline bears the least similarity to other baselines (that describe other behaviors), as it should have learned unique local patterns pertaining to the application. We define the similarity between two sparse representation dictionaries $\delta(\mathbf{D}_i, \mathbf{D}_j)$ as the minimum distance separating the local patterns (atoms) of each dictionary.

$$\delta_{ij} = \min_{D_i D_j} \|d_a - d_b\|_2 \ \forall a, b \leq m \tag{6}$$

where d_a and d_b are local patterns in D_i and D_j, respectively. Our similarity metric is a conservative metric: just one similar set of local patterns would lead us to consider the dictionaries similar. We trained dictionaries for all the profiled applications, while varying the sparsity (sp), number of local patterns to be learned (m), and number of iterations (k). Then, we computed δ_{ij} for all pairs of dictionaries. We consider dictionaries with $\delta_{ij} \leq 0.01$ similar. Figure 5 shows the similarity measure between the dictionaries; each element $< i, j >$ in the result matrix represents the distance measure δ_{ij}. If the distance is more than 0.01, the cell is filled with a white color; otherwise, it is filled in black. The diagonal is white as it refers to δ_{ii} as it compares a dictionary to itself. Most of the elements in the result matrix are black, and thus the learned baselines are different. We used the learned baselines for the rest of the experiments.

2) Comparing Execution Traces: For each application we picked a behavioral trace and computed the anomaly score (LSE) against all the behavioral baselines. For detection we compared the anomaly score to the 95th percentile threshold (λ_{LSE}) from the training data. Figure 6 shows a sample of the LSE anomaly scores of `mysql.exe` compared to the behavior of `vlc.exe` using the behavioral baseline of `vlc.exe`. The error of the `mysql.exe` behavior trace is consistently greater than that of `vlc.exe`. The red reference line in the plot is the detection cutoff. For each behavior baseline we average the true positive and the false positives of detection against all the application traces. Table II shows the true positive and the false positives for each application. The results show a consistently low false positive rate. That is, the baseline does not mark normal behavior as anomalous. On the other hand, the detector is capable of accurately discriminating between applications, with varying degrees. In the case of `mysql`, the accuracy is lower than the others; the reason is that the behavior of a database application has similarities to that of the other applications.

TABLE II
COMPARING EXECUTION TRACES

Application	True Positive	False Positive
vlc.exe	0.9917	0.0483
svchost.exe	0.9932	0.115
explorer.exe	0.9482	0.0076
httpd	0.7966	0.0468
mysqld	0.6425	0.0514

B. Experiment 2: Injecting Attack Behavior

In the second set of experiments, we wanted to evaluate the effectiveness of the baselines in detecting anomalous malicious behavior. We considered two classes of attacks: one-time arbitrary code execution through shellcodes, and permanent code injection. Permanent code injection is used to hide malicious activity within "trusted" applications. Malware families such as Duqu and Dyre use calls such as `ZwOpenThread`, `ZwQueueApcThread`, and `ZwCreateSection` to inject malicious code into Windows subsystem processes. The goal is for KOBRA to be able to detect both permanent and one-time anomalies. In order to evaluate the anomaly detection, we wove malicious behavior into normal execution traces of a process. In the following we explain our weaving process and show the accuracy of the anomaly detection method.

1) Malicious Behavior Weaving: Given a trace of malicious (malware) behavior, we wove the behavior trace into a normal process behavior trace. Weaving of the traces is a reasonable way to emulate malware behavior within a process, because the execution of the exploit happens within the compromised process, and thus the collected trace will reflect the behavior of the exploit. We do not need to prove that the applications are vulnerable, as we do not use specific vulnerabilities; instead, we look at the behavior after the exploit has been "executed." Moreover, weaving malicious behavior instead of

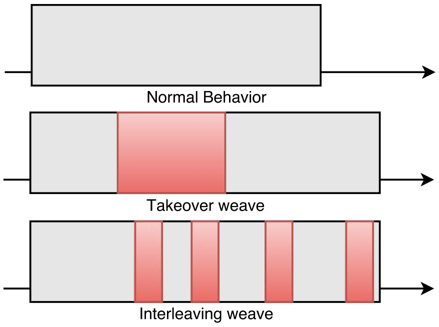


Fig. 7. Malware behavior weaving modes

finding vulnerable versions of applications allows the results to be reproducible by other researchers.

We consider two cases of malicious behavior: takeover and interleaving. In the firs, malware takes over process execution by means of shellcode execution, DLL hijacking, and portable executable (PE) injection into the process image. In the second, the malicious behavior is interleaved with normal behavior; the malware achieves this when it adds a thread to the process execution.

We performed malware weaving by emulating malware behavior and by extracting behavior traces using KOBRA. We added the behavior traces of the malware to the normal behavior of an application. While it might seem that the insertion point should be restricted to network read events, the start of malicious behavior might not align with the network read events, perhaps because of multithreading, for example. Because of the uncertainty, we selected a random insertion point and repeated the experiment multiple times to increase confidence in the result. Finally, we adjusted the timestamps to be consistent. Weaving was performed before the transformation to a complex signal.

The malware behavior was either inserted to cause a shift of normal behavior or interleaved with random periods within the normal behavior. Figure 7 shows the two weaving modes. The gray box is the normal behavior, and the red boxes are the emulated malware behavior.

We studied shellcode behavior in order to assess malicious behavior. First, we studied common behaviors of shellcodes by surveying the Exploit Database by Offensive Security [13]. For all of the 500 Windows exploit samples, we extracted and simulated the binary shellcode using a shellcode debugger, scdbg [14]. The output of the debugger was the list of service calls executed. We identified two behaviors to study:

Reverse Shell (RS): An attacker starts a new socket, connects to a remote server, creates a new process for a shell, and redirects input/output of the new process to the new socket.

Drive-by-Download (DD): An attacker downloads malware from a remote server and creates a new process that loads the downloaded file.

The selected behaviors are by no means an exhaustive list of possible behaviors, as we cannot predict the behavior of an attacker. However, they provide a good starting point for verifying that attack behavior, unknown to our system (which has not been trained for), is getting flagged as an anomaly relative to the learned baseline. Finally, we created custom implementations of the malicious payloads and ran them on a KOBRA-instrumented machine, after which we wove the behavior traces with the applications we wanted to study. When a shellcode is executed, it might cause a new process to be forked or the current process to crash. We do not consider those scenarios, because we are interested in the behavior change due to the malware in the application itself.

TABLE III
TRUE POSITIVE RATE AND FALSE POSITIVE (FP) RATES

	Reverse Shell (RS)			Drive-by-Download (DD)		
Application	LSE	FP	Original	LSE	FP	Original
vlc.exe	0.9800	0.0466	0.8635	0.8941	0.0585	0.6342
svchost.exe	0.9687	0.0753	0.9001	0.8620	0.0627	0.8362
explorer.exe	0.9826	0.0480	0.8888	0.9214	0.0791	0.8822
httpd	0.9933	0.0244	0.9068	0.9641	0.0352	0.94
mysqld	0.9800	0.0499	0.9010	0.9272	0.0848	0.8665
System	0.9094	0.1092	0.8511	0.9041	0.0371	0.7890
$\bar{x}$	0.969	0.0589	0.8852	0.9121	0.0595	0.8247

2) Detection Results: In this experiment, we tested whether that anomaly detection using our learned behavioral baseline is effective against malicious behavior due to reverse shell (RS) and drive-by-download (DD). After weaving the malicious behavior, we computed the anomaly score for each trace and compared the scores against the thresholds. The threshold is selected as the 95th percentile of the anomaly scores from the training data. We compared our results to a kNN classifier that clusters the original trace information without using the behavioral baseline transformation. Table III shows the true positive rates of detection of both reverse shell and drive-by-download behaviors. For reverse shell behavior, the true positive rate was higher than 0.90 for all applications, while the false positive rate was extremely low (≤ 0.07). The false positive rate is consistent with the threshold we picked, the 95th percentile. The detection method using the untransformed traces had lower true positive rate in detecting the malicious behavior. The improvement in detection while having a low false positive rate is important for a resiliency strategy that uses alerts for response. For drive-by-download behavior, the true positive rate was high for all applications (≥ 0.90) with a low false positive rate. Finally, the detection method using the untransformed traces had lower true positive rate.

C. KOBRA's Performance

We tested our current implementation of KOBRA on a machine running Windows 7. We used a suite of performance benchmarks [15] to evaluate its logging overhead and online operation. During logging mode, the benchmarks ran various CPU, memory, disk, and network tests. The results (Figure 8) show that KOBRA has negligible overhead; it did not exceed 6% for any of the tests. The results in Figure 8 are divided by the targeted subsystem: CPU, graphics, memory, and disk. The

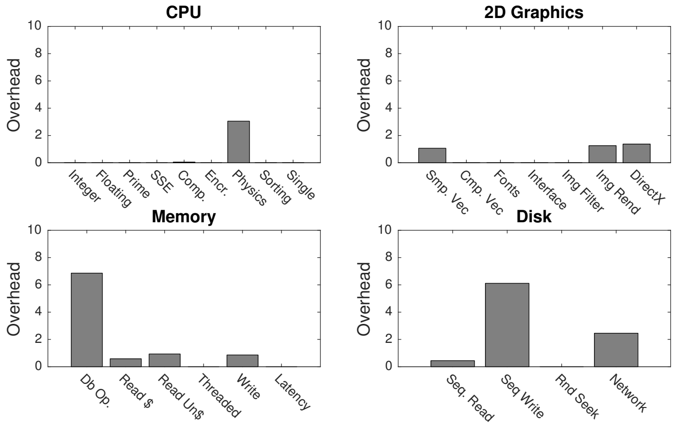


Fig. 8. The overhead due to KOBRA's operations.

low resulting overhead is not surprising; most of the logging functionalities are implemented as callbacks and in-line filters, and the state is updated asynchronously. The 2% network overhead was due to events streaming to the logging server at 277 Kbps.

During online detection mode, the detection algorithm runs with full data collection with logging disabled. That is, KOBRA does not use network communication, but it does increase CPU usage. On average, the batch OMP used for sparse representation runs in 0.12 ms for a batch of 100 signals. We modeled the per-process online anomaly detection as an M/M/1 system. The input to the queue was the behavior trace with 32 elements; the service ran the anomaly detection algorithm 1. The online system is stable when the service rate is higher than the arrival rate, $\frac{\lambda}{\mu} < 1$. Currently, events are coded in 0.12 ms, and the median arrival time per event is 24 ms. Thus the system is stable and is viable for online operation.

V. KOBRA'S IMPLEMENTATION DETAILS

KOBRA is a kernel-monitoring and anomaly detection engine that collects events via collection modules for learning behavioral baselines. It computes the anomaly score and uses algorithm 1 (in Section III-D) to generate an alert when the score exceeds a threshold. The collection modules parse kernel data structures, implement callbacks, and install network and file system filters. The data collected are transferred to the fusion module, which maintains KOBRA's internal state. KOBRA (using the communication module) exports the data to an external logging server for learning. Figure 9 shows the architecture of KOBRA. We implemented KOBRA as a set of drivers for Microsoft's Windows operating system. In this section, we discuss KOBRA's implementation details.

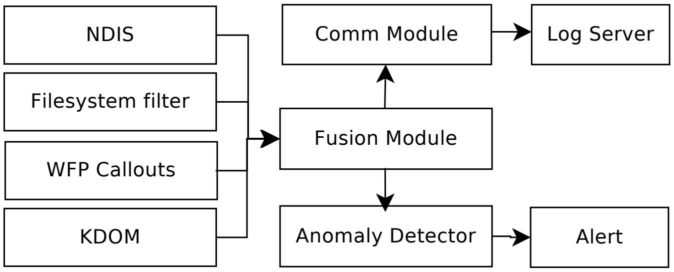


Fig. 9. KOBRA's architecture for logging and anomaly detection.

KOBRA collects events and data (tagged by the originating process) from the kernel to instantiate and maintain an internal state of the host. It employs two strategies for data collection: events-driven collection, and periodic polling. In event-driven collection, KOBRA registers functions to notification and callback objects in the kernel. These functions asynchronously update the internal state to reduce overhead. For periodic polling, it uses worker threads to sample time-varying properties, such as CPU and memory usage. The fusion module maintains a state of the host using data from the collection modules. The state is implemented as a dynamic graph which we encode as a graph stream. Table I lists the events that are currently streamed by KOBRA.

Our implementation uses *only* public APIs to collect kernel events. The Windows operating system (64-bit version) implements kernel patch protection to prevent tampering with the kernel data structure and code. These protections prevent hooking of system calls and interrupt handlers. Even though Kernel Patch Protection (KPP) can be bypassed [9], we limit our collection components to sources that can be accessed without tampering with the operating system. Those include process information, network information, and file system interactions.

1) Process Information: In order to collect information about running processes in the system, we parse the `EPROCESS` structure in the kernel. Windows maintains a list of running processes that contains all the state information pertaining to each process. We register a callback function to get notified when a process is created or terminated. For each process, we compute a secure hash of its image file. The secure hash, instead of the image file, is used to identify the running application; we assign to each process a unique random identifier, KID, instead of using the internal process identifier (PID), which is likely to be reused by the kernel. Each time a process is created or terminated, KOBRA emits an event with a time stamp, KID, parent ID, application ID, and process name. The timestamp is a high-resolution counter ($< 1\ \mu$s) obtained by `KeQueryPerformanceCounter`. The short interval between ticks allows KOBRA to keep an order of the events without using a global synchronized variable.

2) Communication Information: We implemented two kernel components to capture network activities and map the activity to the processes that perform them. A Network Driver Interface Specification (NDIS) filter and a set of Windows Filtering Platform (WFP) kernel callouts listen for incoming and outgoing packets. The NDIS filter uses the destination port, IP address, and protocol tuple to keep track of all remote destinations. The WFP callouts associate a context with each packet in a network flow. The context is the KID of the process that initiated the connection. Two types of events are emitted as a result of communication information: established connection activities and detailed network activities. Those events, along with the sizes of the associated communications (in bytes) are tagged with the remote destination tuples.

3) Storage Information: Storage information is collected by a file system filter (in the file system stack) that listens to file open, read, write, and query operations. The file is uniquely identified by location and name. Each file operation is tagged by the process that initiated the operation and is added to the internal state maintained by the fusion module. A file system filter allows us to capture processes through pipes.

A. Anomaly Detector

The anomaly detector implements Algorithm 1. Learned application behavioral baselines are stored in memory (loaded from a file) and indexed by the secure hash of the application's image file. The secure hash is used to match the process's application with the learned baseline. In step 1, KOBRA separates the behavior into different traces per process. If KOBRA is in learning mode, the traces are exported to an external server. The server learns the behavioral baseline. If KOBRA is in online detection mode, each event in the trace is converted to a complex value. In step 2, batched OMP computes the sparse representation y of a set of the converted traces x. In step 3, the sparse representation error, δ_{SRE}, is computed. An alert is issued if $\delta_{SRE} \geq \lambda_{SRE}$, where λ_{SRE} is the 95th percentile of the SRE from the training data. Otherwise, in step 4, the latent semantic representation approximation of y is calculated. In step 5, the latent semantic error, δ_{LSE}, is calculated. In step 6, an alert is issued if $\delta_{LSE} \geq \lambda_{LSE}$, where λ_{LSE} is the 95th percentile of the LSE from the training data.

VI. DISCUSSION

In our transformation, some semantics of operations are lost, such as file names and IP addresses. Attackers can fake those semantics, a common pitfall of signature-based approaches that renders them ineffective. Even though we capture only file system and network activity, our baselines capture high-level and subtle behavioral differences in the studied applications. The high-level differences emerge from the intended purpose of the application (e.g. a Web server mainly sends network traffic, or a browser mainly receives traffic). Moreover, our baselines capture implementation details in an application, e.g., the size of the buffers used to read files, interleaving between file and network operations, and thread sleep duration. Our approach has certain limitations that we plan to study. We assume that KOBRA is running in a secure manner; while such assumptions are common for intrusion detection systems, we plan to study methods to secure KOBRA's execution or at least detect tampering with it. Moreover, we currently support one mode of operation per application; we plan to study methods to support multiple modes of operation via multiple dictionaries.

VII. RELATED WORK

Anomaly Detection: The techniques for host-based anomaly detection can be classified according to the collected data, the extracted features, and the detection methods.

a) Data Collected: Collection of system calls and function calls in modern operating systems is not possible without reducing the security stature of the OS [16]. In our work, instead of extracting a subset of the features in the network and file activity traces (e.g., bandwidth), we collect continuous file and network activity and use the whole trace for analysis. Forest [17], [18] collected system calls; Peiser [19] collected function calls; Tang et al. [20] collected architectural information, e.g., cache misses; Malone et al. [21] collected hardware performance counters, e.g., INS; and, finally, Gao et al. [22] collected "gray-box" measurements.

b) Features: After collection, monitoring data features are extracted for anomaly detection. Several approaches have been proposed in which data are arranged into: 1) short sequences with events substituted into natural numbers [17] (n-grams), 2) frequency and wavelet transformation coefficients [23], 3) entropy values, or 4) Fisher scores [20]. Selected features should discriminate between normal and anomalous behavior. In our work, the features consist of the decomposition of the traces over the learned set of basis vectors, as opposed to designed features. Thus, our features are always well-suited for the supplied data.

c) Analysis Method: Finally, the selected features are used to learn normal behaviors; we refer the readers to the extensive surveys by Agrawal et al. [24] and Chandola et al. [25] on anomaly detection. On top of the surveys, some researchers use Markov models [26], [27] or finite state machines [28] to learn relationships between operations. Such methods do not take into account delays and call semantics. PCA methods have gained popularity [29]. They transform the data into independent components for clustering; PCA only considers second-order statistics, unlike to learned dictionaries which exploit the data beyond variance.

Monitoring Software: Dunlap et al. [30] argue that the logging capabilities of the kernel are not trustworthy and moved logging to a hypervisor. They record all events that occur in a guest including, CPU counters, network messages, file I/O, and interaction with the peripherals. OSck [31] implements rootkit protection by monitoring kernel integrity. The trust argument for hypervisor monitoring has been weakened by multiple compromises from hardware below [32] and from guest machines above [33]. The semantics gained from monitoring in the kernel instead of the hypervisor outweighs the (already weakened) trust argument. OSSEC [34] and AIDE [35] provide kernel-level monitoring of the registry and file integrity. These tools do not provide process-tagged activity for behavior analysis.

VIII. CONCLUSION

We proposed a method to model application behavioral baselines using practically monitored data from the kernel. While typical anomaly detection methods assume access to low-level system calls, in this work, we use file and network activity captured by KOBRA, a kernel-monitoring and anomaly detection engine. We propose a novel transformation from KOBRA's data to a complex-valued discrete-time signal. The signals are used to learn a sparse representation dictionary and a latent semantic analysis transformation that serve as a behavioral baseline for each application. The generated baseline captures the uniqueness of an application. Our approach is effective in detecting simulated attacks with a low false positive rate. Moreover, KOBRA has low overhead and is stable for online operation. In the future, we plan to expand our experiments to study more attack behavior. Moreover, we plan to study online dictionary learning to help KOBRA learn baselines. Finally, we plan to design host-level response mechanisms to work towards intrusion resilience.

ACKNOWLEDGMENT

The authors would like to thank Jenny Applequist for editing this paper. They also thank the reviewers for their constructive feedback. This material is based upon work supported by the Department of Energy under Award Number DE-OE000078.

REFERENCES

[1] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE Document, Tech. Rep. MTR-2547, Vol. 1, 1973.

[2] E. Dijkstra, “The structure of the the multiprogramming system,” in *Classic Operating Systems*, P. Hansen, Ed. Springer New York, 2001, pp. 223–236. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-3510-9_12

[3] S. Engle, S. Whalen, and M. Bishop, “Modeling computer insecurity,” Dept. of Computer Science, University of California at Davis, Davis, CA 95616-8562, Tech. Rep. CSE-2008-14, 2008.

[4] R. Oppliger and R. Rytz, “Does trusted computing remedy computer security problems?” *IEEE Security Privacy*, vol. 3, no. 2, pp. 16–19, March 2005.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[6] J. McHugh, “Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory.”

[7] 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.

[8] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*, 6th ed. Addison-Wesley Professional, August 2005.

[9] A. Allievi, “The Windows 8.1 kernel patch protection,” http://vrt-blog.snort.org/2014/08/the-windows-81-kernel-patch-protection.html, 2014.

[10] C. Parampalli, R. Sekar, and R. Johnson, “A practical mimicry attack against powerful system-call monitors,” in *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS ’08. New York, NY, USA: ACM, 2008, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/1368310.1368334

[11] Y. Pati, R. Rezaifar, and P. Krishnaprasad, “Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition,” in *1993 Conference Record of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*, Nov. 1993, pp. 40–44.

[12] M. Aharon, M. Elad, and A. Bruckstein, “K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation,” *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311–4322, Nov. 2006.

[13] “Exploit database,” http://exploit-db.com.

[14] D. Zimmer, “scdbg,” https://github.com/dzzie/VS_LIBEMU, 2011.

[15] “Performancetest 8.0,” http://www.passmark.com/products/pt.htm.

[16] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals, Part 1*, 6th ed., ser. Developer Series. Microsoft Press, September 2012.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, Aug. 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298081.1298084

[18] G. Creech and J. Hu, “A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns,” *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 807–819, April 2014.

[19] S. Peisert, M. Bishop, S. Karin, and K. Marzullo, “Analysis of computer intrusions using sequences of function calls,” *IEEE Trans. Dependable Secur. Comput.*, vol. 4, no. 2, pp. 137–150, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2007.1003

[20] A. Tang, S. Sethumadhavan, and S. Stolfo, “Unsupervised anomaly-based malware detection using hardware features,” in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer International Publishing, 2014, vol. 8688, pp. 109–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11379-1_6

[21] C. Malone, M. Zahran, and R. Karri, “Are hardware performance counters a cost effective way for integrity checking of programs,” in *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, ser. STC ’11. New York, NY, USA: ACM, 2011, pp. 71–76. [Online]. Available: http://doi.acm.org/10.1145/2046582.2046596

[22] D. Gao, M. K. Reiter, and D. Song, “On gray-box program tracking for anomaly detection,” in *Proceedings of the 13th Conference on USENIX Security Symposium*, ser. SSYM’04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251383

[23] W. Lu and A. A. Ghorbani, “Network anomaly detection based on wavelet analysis,” *EURASIP J. Adv. Signal Process*, pp. 4:1–4:16, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/837601

[24] L. Ding, C. Pang, L. M. Kew, L. C. Jain, R. J. Howlett, S. Agrawal, and J. Agrawal, “Survey on anomaly detection using data mining techniques,” *Procedia Computer Science*, vol. 60, pp. 708–713, 2015.

[25] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009.

[26] S.-B. Cho and H.-J. Park, “Efficient anomaly detection by modeling privilege flows using hidden Markov model,” *Computers & Security*, vol. 22, no. 1, pp. 45–55, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404803001123

[27] D. Gao, M. K. Reiter, and D. Song, “Behavioral distance measurement using hidden Markov models,” in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, D. Zamboni and C. Kruegel, Eds. Springer Berlin Heidelberg, 2006, vol. 4219, pp. 19–40. [Online]. Available: http://dx.doi.org/10.1007/11856214_2

[28] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, “A fast automaton-based method for detecting anomalous program behaviors,” in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 144–155.

[29] W. Wang, X. Guan, and X. Zhang, “A novel intrusion detection method based on principal component analysis in computer security,” in *Advances in Neural Networks - ISNN 2004*, ser. Lecture Notes in Computer Science, F.-L. Yin, J. Wang, and C. Guo, Eds. Springer Berlin Heidelberg, 2004, vol. 3174, pp. 657–662.

[30] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 211–224, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844148

[31] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, “Ensuring operating system kernel integrity with OSck,” *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 279–290, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961295.1950398

[32] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin, “Attacking hypervisors via firmware and hardware.” Blackhat 2015.

[33] R. Wojtczuk, “Poacher turned gamekeeper: Lessons learned from eight years of breaking hypervisors.” Bromium Endpoint Protection, July 2014.

[34] A. Hay, D. Cid, and R. Bray, *OSSEC Host-Based Intrusion Detection Guide*. Syngress Publishing, 2008.

[35] R. R. Lehti, P. P. Virolainen, and R. van den Berg, “Aide (advanced intrusion detection environment),” http://sourceforge.net/projects/aide, 2013.