A Trace Abstraction Approach for Host-based Anomaly Detection

[1]Syed Shariyar Murtaza, [1]Wael Khreich, [1]Abdelwahab Hamou-Lhadj, [2]Stephane Gagnon
[1]Software Behaviour Analysis (SBA) Research Lab, Department of Electrical and Computer Engineering,
Concordia University, Montreal, QC, Canada
[2]Département des sciences administratives
Université du Québec en Outaouais
[1]{smurtaza, wkhreich, abdelw}@ece.concordia.ca, [2]stephane.gagnon@uqo.ca

***Abstract*— High false alarm rates and execution times are among the key issues in host-based anomaly detection systems. In this paper, we investigate the use of trace abstraction techniques for reducing the execution time of anomaly detectors while keeping the same accuracy. The key idea is to represent system call traces as traces of kernel module interactions and use the resulting abstract traces as input to known anomaly detection techniques, such as STIDE (the Sequence Time-Delay Embedding) and HMM (Hidden Markov Models). We performed experiments on three datasets, namely, the traditional UNM dataset as well as two modern datasets, Firefox and ADFA. The results show that kernel module traces can lead to similar or fewer false alarms and considerably smaller execution times compared to raw system call traces for host-based anomaly detection systems.**

Keywords—Host-based Anomaly Detection System, Trace Analysis andAbstraction, System Call Traces, Software Security, Software Dependability

I. INTRODUCTION

Host-based anomaly detection systems have gained popularity in recent years due to the increasing number of security threats. Unlike misuse-based detection systems which focus on detecting known attacks signatures, anomaly detection systems can detect both known and unknown attacks. The common approach is to use system call traces to model the normal behavior of the system, since system calls are the gateway between user and kernel spaces [*35*] [*8*]. The resulting model can then be used as a baseline for the detection of abnormal behavior during operation.

Despite considerable efforts in the field, anomaly detection systems still suffer from high false alarm rates (see [*32*] [*38*] [*35*] [*8*] [*12*]). In addition, the execution time (including training, validating, and testing) of anomaly detection techniques can be quite high. For example, training the Hidden Markov Model (HMM) on large normal system traces [*35*] [*4*] [*11*] [*12*] may take days. This, by itself, is not an issue if the system needs to be trained only once. However, in practice, because of new patches and modifications to the system, it is common to train the system multiple times to adapt it to the new changes. Therefore, approaches that can reduce the execution time and the false positive rate while keeping the same detection accuracy are needed.

After careful examination of different attacks found in publicly available Linux-based datasets (e.g., UNM [*8*] [*31*], ADFA [*6*]), we have noticed that attacks tend to crosscut different modules of the Linux operating system, such as, file system, architecture (arch), core kernel, memory management, etc. For example, to execute an arbitrary code in the system, an attacker might execute the system call "sys_execve" in the arch module, call "sys_brk" in the memory management module to increase the allocated memory, and manipulate system files using the system calls in the file system module.

These observations have led us to investigate the use of OS kernel modules for the detection of anomalies. More precisely, we propose an approach for anomaly detection in which we transform the content of system call traces into traces of kernel modules by replacing each system call by the kernel module that defines it. Since the number of kernel modules (eight in a typical Linux distribution) is considerably less than the number of distinct system calls (around 300), this trace abstraction technique results in fewer distinct events, which should yield significant reduction in the time it takes to train and test anomaly detectors. As we will show in the case study, this trace abstraction method also results in similar, and sometimes better, accuracy when compared to the use of raw system call traces.

In this paper, we choose to use two anomaly detection techniques: the Sequence Time-Delay Embedding (STIDE) [*8*] and the Hidden Markov Model (HMM) [*35*] [*34*]. We selected these two techniques because they have shown to produce best results [*8*] [*35*] [*34*]. We supplied these detection techniques with traces of kernel modules extracted from three publicly available system call based datasets. The first one is the University of New Mexico (UNM) dataset [*8*] [*31*]. Although the system call datasets from UNM have been criticized in related literature [*29*], they are still being used for benchmarking anomaly detection systems due to the lack of many publicly available datasets [*2*] [*5*]. The two other datasets consist of a newly developed Firefox dataset [*22*], and the ADFA Linux dataset [*6*]. We created the Firefox trace dataset by using contemporary software testing and hacking techniques on Linux [*22*]. The ADFA dataset is also a contemporary dataset for Linux, created by researchers at the University of New South Wales, Australia [13].

The results show that the use of kernel module traces reduces significantly the execution time of the selected anomaly detectors without necessarily compromising their accuracy. In fact, it resulted in better false positive rates in the case of STIDE, and almost similar accuracy when used with HMM.

The rest of the paper is organized as follows: Section II presents a brief background and a literature review; Section III describes the methodology and the evaluation measures; Section IV presents the case study on datasets; Section V shows the threats to validity; and Section VI concludes with future work.

II. Background and Related Work

Traditional Intrusion Detection Systems (IDS) rely on misuse detection, which compare software behavior with a database of known attributes extracted from known attacks. When a pattern of attack is found, the behavior is considered as anomalous. Another type of intrusion detection systems, the focus of this paper, works by building a robust baseline of the normal behavior of a system and then monitors for deviations from this baseline during system operation [*25*]. These systems are called anomaly detection systems.

Anomaly detection systems can be classified into Host-based Intrusion Detection Systems (HIDS) or Network-based Intrusion Detection Systems (NIDS). NIDS examine network traffic to detect anomalies; e.g., the use of Bayesian network on network traffic records to detect anomalies [*32*] and the extraction of rules by mining tcp-dump data to detect anomalies [*27*]. HIDS focus on using metrics present in a host system to detect anomalies. A type of HIDS uses different algorithms on normal audit records (logs) of a host (e.g., CPU usage, process id, user id, etc.). These systems measure an anomaly threshold and raise alerts when particular attribute values of a new record are above the threshold. For example, using multivariate statistical analysis on audit records to identify anomalies [*37*], and using frequency distribution based anomaly detection on shell command logs [*38*]. Another type of HIDS trains different algorithms on system calls of normal software behavior. These systems raise alerts when the deviation from normal system calls is observed in unknown software behavior (e.g., a trace). Anomaly-based HIDS focusing on system calls deviations are related to our work and are described below.

Forrest et al. [*8*] propose STIDE (Sequence Time-Delay Embedding), an anomaly detection system based on sequences of system calls. STIDE builds a database of normal sequences by sliding a window of length ‘n’ on normal traces. Sequences of similar length ‘n’ are also extracted from traces of unknown behavior and compared with the database. If an unknown sequence is found in a trace then it is considered as anomalous. Hofmeyr et al. [*13*] improve the binary decision of STIDE by computing the Hamming distance between two sequences to determine how much one sequence differs from another. Warrender et al. [*35*] use a locality frame count (i.e., the number of mismatches during the last ‘m>n’ calls) on the output of STIDE instead of the Hamming distance to compute the anomalous signal of an attack. Furthermore, Warrender et al. compare the performance of several anomaly detectors [*35*], among these STIDE and HMMs have shown the best performance on UNM datasets.

When Warrender et al. [*35*] train HMM on system call sequences, they raise alerts as the probability of a system call in a sequence goes below a certain threshold. On the other hand, Wang et al. [*34*] train HMM on normal system call sequences and raise alerts when the probability of a whole sequence of system calls is below a threshold rather than individual calls in a sequence [*35*]. Similarly, Yeung and Ding [*38*] also employ HMM on system call sequences, and refer to this approach as dynamic modelling. Yeung and Ding [*38*] also measure frequency distributions for shell command logs and called it static modelling. They show that dynamic modelling performs better than static modelling. Other researchers, Cho and Park [*4*], use HMM to model using system calls the execution of only normal root privilege acquisitions. This allows them to detect 90% of the illegal privilege flow attack. Hoang et al. [*11*] move a step ahead and propose a multiple layer detection approach in which one layer train the STIDE on system calls and another layer train HMM on system calls. They combine the output of both to detect anomalies. Hoang et al. [*12*] also improve their earlier work [*11*] by combining HMM and the STIDE using fuzzy inference engine. Although HMM has been commonly and successfully used in related work on system call anomaly detection, its large training time remains an issue. Researchers like Hu et al. [*14*] propose an incremental HMM training technique to reduce its training time by 50%.

A large number of machine learning techniques have also been proposed for detecting anomalous system calls. For instance, researchers have employed standard multilayer perceptron [*1*], Elman recurrent neural network [*10*], self-organizing maps neural network [*18*], and radial basis functions based neural networks [*2*]. SVM, decision trees, and K-means clustering have also been used on system calls extracted from static analysis [*39*]. The researchers in [*1*] and [*39*] employed training on both normal and anomalous traces to detect anomalies.

Jiang et al. [*15*] extract varied length n-grams from call traces of normal behavior, and build an automaton that represents the generalized state of the normal behavior: they use this automaton to detect anomalous behavior in traces. Tandon [*28*] propose different variations of LEARD, a conditional rule learning algorithm [*24*], to learn rules with sequences of system calls and their arguments. They also propose to generate new rules for the rules pruned due to false alarms on validation data. They argue that new rule generation increases detection rate and coverage on a limited training set. Several other researchers also consider the use of system calls and arguments to strengthen HIDS against the control flow, data flow, and mimicry attacks [*17*] [*20*] [*3*] [*19*]. However, these techniques remain constrained by selecting key system calls and arguments due to multitude of argument values.

Recently, Creech and Hu [5] propose to generate seen and unseen patterns of system calls from normal traces by using production rules of Context Free Grammar (CFG). They also propose a semantic rule that occurrences of patterns in normal traces are greater than anomalous traces and train ELM neural network on occurrences of patterns. They claim that the approach is applicable on multiple processes simultaneously but recognize that the accuracy will be higher on individual processes. This is because false alarm rate reach 20% (out of 4373 normal traces) in one dataset. They also claim portability across version of different operating systems but the results show up to 100% false positive rate for 80% of the attacks.

In prior work, we proposed a method to automatically differentiate between normal and anomalous traces of kernel states [*22*]. Kernel state modeling detects anomalies by measuring deviation between normal and anomalous traces of kernel modules. In this paper, we investigate the effect of kernel modules in reducing the false positives and execution time of the existing well-known anomaly detection techniques, such as the STIDE and HMM. To the best of our knowledge, there has been no such work to evaluate HMM and STIDE on traces of kernel modules for the detection of software anomalies.

III. Methodology

Figure 1 shows the steps of our approach for detecting anomalies using kernel module traces instead of raw system call traces. We first transform the traces into kernel module traces. Then, we train the anomaly detection techniques on the normal kernel module traces. During training, an anomaly detection technique develops a model of traces. The next step is to validate the model on another set of normal traces. During validation, a technique may (or may not) adjust decision thresholds of its model to lower its false alarm rate. Finally, the model is used to evaluate traces coming from a system in operation to detect anomalies (testing phase). The best model is the one which has a low false alarm rate and a high true positive rate when applied to test traces.

A. Transforming System Call Traces into Kernel Module Traces

A system call trace is actually a temporally ordered collection of system calls. Examples of system calls include sys_open, sys_read, and sys_close, which open, read, and close a given file (specified in the argument list). To generate a trace of kernel modules from a trace of system calls, we replace each system call with the kernel module in which it is defined. Table I shows the list of modules and the total number of system calls that belong to each module[1]. For example, the File System (FS) module contains 131 system calls (i.e., sys_open, sys_read, sys_close, etc.). There are a total of eight distinct modules in a typical distribution of Linux.

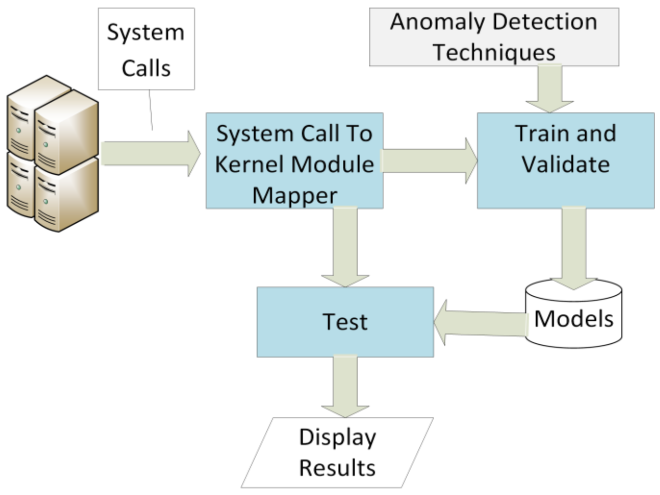
System Calls
Anomaly Detection Techniques
System Call To Kernel Module Mapper
Train and Validate
Test
Models
Display Results

Figure 1. Overview of the proposed approach

Note that although this study is tuned towards Linux machines, it can easily be adapted to other operating systems (OS) given that the mapping between the OS modules and system calls is provided.

Table I MAPPING BETWEEN SYSTEM MODULES AND SYSTEM CALLS

Module	**Module in Linux Source Code**	**# of System Calls**
AC	Architecture	10
FS	File System	131
IPC	Inter Process Communications	7
KL	Kernel	127
MM	Memory Management	21
NT	Networking	2
SC	Security	3
UN	Unknown	37

The intuition behind using kernel module interaction to detect anomalies is that, to be effective, attackers must request different OS services including accesses to memory, files, and network. Our hypothesis is that such attacks can be detected by simply observing the interactions among kernel modules. If the hypothesis is valid then this higher level abstraction can help built anomaly detectors that are not only accurate but also scalable by reducing their execution time (training, validation, and testing) as well as storage space required to save normal models generated from large systems.

We recognize, however, that not all attacks can be detected at this level of abstraction. There might be situations where an attack manifests itself through system call sequences that occur only within the same kernel module.

[1] The mapping of modules and system calls for Linux can be found at http://syscalls.kernelgrok.com and is applicable to a 32 bit kernel 2.6.35

We conjecture that such attacks may have a lesser effect that those that combine multiple OS services. Nevertheless, we anticipate that a tool that supports our approach should provide a mechanism to vary the level of abstraction as needed based, for example, on security risks and the level of security required. In this case, the main gain in terms of execution time is the time it takes to detect attacks in operation.

B. *Anomaly Detection Techniques*

As mentioned earlier, we have selected STIDE and HMM, since they are among the most commonly used and successful system call anomaly detectors, to assess the impact of kernel module traces on the detection accuracy and the execution time.

STIDE works by extracting sequences of length 'n' from a trace by sliding a window one event (e.g., system call) at a time [*8*] [*13*] [*35*]. For example, for a trace having system calls "3, 6, 195, 195", two sequences "3, 6, 195" and "6, 195, 195" of length 3 can be extracted. STIDE extracts sequences from normal traces and then compares them against the sequences in an unknown trace. If a new sequence is found in an unknown trace then it is considered as anomalous. The Hamming distance between sequences can be used to adjust the decision threshold to reduce false alarms. For example, using exact matching the sequence "3, 5, 195" is considered as anomalous even though the mismatch occurs only at one position—i.e., a hamming distance of only one. If the minimum Hamming distance matching criterion is set to more than one, then it is considered as a normal sequence.

HMM is a stochastic model for sequential data and hence it is naturally suitable for modeling temporal order of system call sequences [*26*]. The process is determined by a latent Markov chain having a finite number of states, N, and a set of output observation probability distributions, B, associated with each state. Starting from an initial state N_0, the process transits from one state to another according to the matrix of transition probability distribution, A, and then emits an observation symbol O_k from a finite alphabet (i.e., M distinct observable events) according to the output probability distribution, $B_j(O_k)$, of the current state Nj. HMM is typically parameterized by the initial state distribution probabilities (Π), output (emission) probabilities (B), and state transition probabilities (A). Baum-Welch algorithm is used to train the model parameters to fit the sequences of observations, T [*26*]. During the validation phase, HMM adjusts the decision threshold (log likelihood) of prediction of anomalous alarms on T sequences from traces. In the testing phase, if the probability value of any sequence in a trace is below the selected threshold, then we consider the trace as anomalous otherwise we consider it as normal.

C. *Evaluation Criteria*

First, we evaluate the accuracy of an anomaly detection technique using the true positive rate (TP) and false positive rate (FP) measures. True positive (or detection) rate is measured by Equation 1. Similarly, false positive (or false alarm) rate is measured by Equation 2.

$$TP = \frac{Number\ of\ detected\ attacks\ (anomalies)}{Total\ number\ of\ attacks\ (anomalies)} \times 100$$

Equation 1. True positive rate

$$FP = \frac{\begin{array}{c}Number\ of\ normal\ traces\ detected\\ as\ anomalous\end{array}}{Total\ number\ of\ normal\ traces} \times 100$$

Equation 2. False positive rate

Second, we evaluate the anomaly detection technique by measuring its execution time. The execution time is measured *by accumulating the training time, validation time and testing time* on the traces for the anomaly detection technique.

We measure the TP rate, FP rate and the execution time for both system call traces and kernel module traces and for every anomaly detection technique. We then compare the measures to determine the differences in the use of the two types of traces.

IV. CASE STUDY

The objective of the case study is to address the following research questions:

(RQ1) What is the effect on accuracy when an anomaly detection technique is trained on kernel module traces instead of raw system call traces?

(RQ2) What is the effect on the execution time when an anomaly detection technique is trained on kernel module traces instead of system calls?

A. *Datasets*

We have used three different system call datasets: University of New Mexico (UNM) dataset [*31*], Firefox dataset [*22*], and ADFA Linux dataset [*6*]. UNM dataset is a common benchmark for system call anomaly detection ([*35*] [*13*] [*12*] [*38*] [*34*] [*11*]). More information about the UNM dataset can be found in [*31*]. We have selected from UNM datasets the programs (Stide[2] and Xlock) that are executed on Linux and have a large number of traces for training and testing. The number of traces used in Stide and Xlock are presented in Table II. We have divided the normal traces randomly into three parts: training, validation, and testing as required by the techniques.

Table II DESCRIPTION OF THE SUBJECT DATASETS

Program	# Normal Traces			#Attack Types	#Attack Traces
	Training	Validation	Testing		
Stide	400	200	13126	1	105
Xlock	91	30	1610	1	2

[2]Stide is a dataset of system call traces collected from the STIDE anomaly detector itself. We will use Stide to refer to the dataset and STIDE for the anomaly detector.

Firefox	125	75	500	5	19
ADFA	833	300	4073	60	686

Each trace for a program in the UNM datasets is a sequence of system calls generated from the execution of one process. A process corresponds to a task or a sub-task that is fulfilled by that program. Xlock and Stide include several traces for one attack (attack type in Table II). The Xlock program is intruded by a buffer overflow attack through one of its command line options. The attack on the Stide anomaly detector is a denial of service attack that affects the memory request of any program in execution. If an anomaly is detected in any of the traces of an attack, we consider that an attack has been detected.

UNM datasets are still used as one of the main benchmarks for anomaly detection systems but they are more than a decade old. Moreover, the normal traces in UNM datasets are collected by executing small programs for a longer period of time which resulted in the same system call execution paths. To overcome this, we have selected two additional datasets, namely the Firefox [*22*] and ADFA Linux [*6*] datasets.

We created the Firefox dataset by collecting traces of normal behavior for Firefox 3.5 by executing seven different testing frameworks (test suites) [*22*]. Each test framework executed different components and functionalities of Firefox. We determined the completeness of normal behavior of Firefox by measuring its code coverage for test case executions. The execution of seven different test suites resulted into approximately 60% source code coverage and 5931 passing test case files corresponding to approximately 1.3 TB of traces. Due to such a large set of trace dataset, we randomly selected equal number of traces from each test suite. We also separated each trace into traces of individual processes of Firefox. Table II shows the number of per-process traces of normal and anomalous traces of Firefox.

We collected anomalous traces on Firefox by launching contemporary attacks against Firefox, selected from public advisories [*23*] and public resources, such as Metasploit [*21*]. We executed five different attacks on Firefox and collected their corresponding traces. The first attack was a memory corruption exploit that tried to execute an arbitrary code. The second attack was an integer overflow attack that caused a denial of service and executed an arbitrary code. The third attack manipulated dangling pointers in the tree data structure of Firefox causing an arbitrary code execution and denial of service. The fourth attack was a DOM exploit causing memory corruption and the fifth attack was a null pointer exploit causing denial of service. The number of attacks and corresponding traces are shown in Table II. The Firefox dataset can be downloaded from our website: http://www.ece.concordia.ca/~abdelw/sba/FirefoxDS.

ADFA Linux dataset is publicly available on the site of University of New South Wales, Australia[3] [*5*]. ADFA dataset was generated on a fully patched Ubuntu Linux 11.04 operating system with Apache 2.2.17 web server, PHP 5.3.5 server side scripting engine, TikiWiki 8.1 content management system, FTP server, MySQL 14.14 database management system and an SSH server. This configuration was chosen to represent a realistic modern target and publicly known security vulnerabilities were exploited in them [*23*]. Normal dataset of traces was collected by executing different activities, such as web browsing, Latex document preparation and etc. Each normal trace consists of sequences of system calls occurred during normal system activities. The traces in ADFA dataset contain sequences of system calls of all the processes and not separated on a per-process basis. The ADFA dataset is described in Table II.

[3] "www.cybersecurity.unsw.adfa.edu.au/ADFA IDS Datasets/"

In the ADFA dataset, anomalous traces were collected by attacking a system using a certified penetration tester. Sixty different attacks, belonging to six types of attack vectors, were executed on the system by using modern penetration testing tool like Metasploit [*21*]. These attacks included web-based exploitation, simulated social engineering, poisoned executable, remotely triggered vulnerabilities, remote password brute force attacks and system manipulation using the C100 webshell. Each attack consists of multiple traces collected during the execution of attack. If an anomaly is detected in any of the traces of an attack, then we consider that the attack is detected. More detailed information about the ADFA dataset can be found in [*6*].

B. *Measuring Accuracy of Anomaly Detection Techniques*

The results of evaluating STIDE, the first selected anomaly detector, on system call traces and kernel module traces of the datasets of Table II are shown in Table III. We chose to use STIDE with window size five. Other window sizes can also be used. Since it does not require optimization of parameters, STIDE was trained on the traces of both training set and validation set.

Table III shows that when using window of width five, the false positive rate in the case of ADFA has been reduced from 83.37% to 12.69% while keeping the same true positive rate (i.e., 100%). We examined traces of ADFA to understand the root cause of such discrepancy. We found that a model that is based on kernel module traces has a better generalization property than the one based on system calls when used with STIDE. For example, consider the following two sequences S1 and S2:

S1: fork, read, read, fork, read, read, fork, read,

S2: fork, read, read, write, fork, read, read, fork, read

An STIDE-based model that recognizes S1 will raise an alarm when encountering S2 (because of the use of the 'write' system call that is not in S1). We believe that this is too restrictive (note that, for the purpose of illustration, we are taking here as an example a simple implementation of STIDE that uses exact matching only). It is reasonable to assume that the 'write' operation is legitimate because it was preceded and followed by the same sequence of system calls. An illegitimate 'write' will most likely trigger a different type of sequence. At the level of kernel module interactions, S1 and S2 are both represented in the same way: KL, FS, KL, FS, KL, FS (KL is used for the fork system call and FS is used to replace read and write) which reduces the number of false positives.

The result obtained on ADFA is consistent with the one obtained when applying the approach to the Firefox dataset, which is another large dataset. The false positive rate is reduced from 44.6% to 10.6%. Note that Firefox model for normal behavior was generated with high code coverage so as to obtain a representative model. This explains why STIDE (with system calls) had only 40.6% false positive rate compared to ADFA. For smaller systems, Stide and Xlock, kernel module traces outperformed system call traces when applied with STIDE, though the false positive rate in both cases is small (up to 4%), due mainly to the size of these systems.

Table III RESULTS USING STIDE

Dataset	Trace Type	TP rate	FP rate
Window width 5			
ADFA	System Calls	100%	83.37%
	Kernel Modules	100%	12.69%
Xlock	System Calls	100%	1.43%
	Kernel Modules	100%	0.31%
Stide	System Calls	100%	4.97%
	Kernel Modules	100%	0.015%
Firefox	System Calls	100%	44.60%
	Kernel Modules	100%	10.6%

The results of the evaluation of HMM are shown in Table IV. We evaluated HMM with different number of states ($N = \{5,10,15,20\}$). Each trace is segmented into contiguous sequences of length 100 events, using a sliding window shifted by one event. In our previous work on system call anomaly detection [*22*], we empirically found that HMM trained on sequences of length 100 events provide a high level of detection accuracy. To improve the execution time of HMM, we removed the duplicate sequences from all the traces of training, validation and testing set. This reduced the number of sequences by approximately 90%. We then applied HMM on the adjusted traces by implementing it in R [*30*].The number of states selected for each trace type and dataset are also shown in Table IV. Note we deliberately varied the number of states so as to obtain 100% detection accuracy. This tuning of HMM during training is needed for HMM to be effective in operation.

Table IV shows the reduction in the false positive rate achieved by HMMs trained using the kernel modules. In all cases, except for ADFA, we obtained almost the same false positive rate with a 100% true positive rate. This is because, unlike STIDE which relies merely on measuring similarity, HMM uses a probability model that generalizes to new (unseen) sequences.

For ADFA, HMM using kernel modules did not perform as well as with system calls. We think that this is due to the nature of the ADFA dataset itself. The dataset was collected by exercising various scenarios on multiple Linux systems. The number of normal traces used for training HMM is too small (833), which resulted in many unseen cases during operation, hence a high positive rate for both representations (system call and kernel module traces) than in other datasets. We are still not sure as to why kernel module traces resulted in more false positives than system calls. We think that HMM is sensitive to the number of unique alphabets in the dataset (eight in the case of kernel module traces and 57 unique system calls used in ADFA). More investigation is warranted.

Table IV RESULTS USING HIDDEN MARKOV MODEL (HMM)

Dataset	Trace Type	States	TP rate	FP rate
ADFA	System Calls	100	100%	40%
	Kernel Modules	100	100%	60%
Xlock	System Calls	10	100%	0.00%
	Kernel Modules	10	100%	0.00%
Stide	System Calls	100	100%	2.0%
	Kernel Modules	100	100%	1.0%
Firefox	System Calls	50	100%	23.0%
	Kernel Modules	50	100%	23.0%

Despite the results we obtained using HMM on ADFA, we have demonstrated through this case study that kernel module traces when using STIDE perform better than when using system calls sequences. They provide similar results in most cases when using HMM. This answers the first research question (RQ1).

C. *Measuring Execution Time of Anomaly Detection Techniques*

Table V shows the execution time of STIDE and HMM. The execution time shown includes training, validation and testing time. The timing information is collected on a machine having Intel core i5, 8GB RAM and 64 bit Ubuntu 12.04. We implemented STIDE in Java, and evaluated the same implementation on both system call traces and kernel module traces. Similarly, we implemented HMM in R [*30*].

Table V shows the execution time of STIDE with window of width five events. It can be observed that the gain in term of execution time when using STIDE is between 70% (Stide) to 99% (Firefox). In the case of HMM, kernel module representation achieves an execution time gain of up to 96%. Table VI shows the storage space that both representations require. As expected kernel module traces consume less space (up to 93% less), which clearly shows the advantage of to the use of kernel module traces, especially when applied to large (and more realistic) systems.

We can therefore conclude that kernel module traces can be used to significantly reduce the execution time of the anomaly detection techniques. This answers the research question (RQ2).

Table V EXECUTION TIME IN MINUTES INCULDING TRAINING, VALIDATION AND TESTING OF STIDE AND HMM ON SYSTEM CALLS (SC) AND KERNEL MODULES (KM).

	STIDE SC (A)	STIDE KM (B)	Gain (1-B/A)	HMM SC (C)	HMM KM (D)	Gain (1-D/C)
Xlock	9.2	1.11	88%	1.74	0.93	47%
Stide	2.53	0.75	70%	6.32	0.49	92%
Firefox	60.96	0.6	99%	150.6	6.65	96%
ADFA	23.9	1.43	94%	3.97	1.35	66%

Table VI COMPARISON OF TRACE SIZES IN BOTH REPRESENTATION

	Size of SC (A)	Size of KM (B)	Size Gain (1-B/A)
Xlock	47.4	30.3	36%
Stide	37.2	4	89%
Firefox	270.6	18	93%
ADFA	10.9	3	72%

V. THREATS TO VALIDITY

We describe threats to validity in four categories: conclusion validity, internal validity, construct validity, and external validity [*36*].

A threat to conclusion validity exists in the use of only two anomaly detection techniques, HMM and STIDE. It is possible that other techniques might not be able to detect attacks as efficiently as HMM and STIDE on kernel modules. However, this threat is mitigated by the fact that HMM and STIDE are the two techniques which yield best results in prior research. These techniques also detected all the attacks using kernel module traces in this study.

A threat to internal validity exists in the implementation of anomaly detection techniques. We have mitigated this threat by manually verifying the outputs of the techniques on pre-known examples.

A threat to construct validity exists in the use of only system calls as the foundation of kernel modules. Some attacks might go undetected due to the transformation of finer grain events like system calls to higher level modules, which could reduce the sensitivity of detection. In general, it is possible that an attacker might evade detection by crafting an attack using system call sequences that belong to the normal system behavior only, such as the mimicry attacks [*33*]. To capture the manifestation of mimicry attacks, several authors proposed to include additional features such as system call arguments [*20*] [*3*] and return values [*17*]. In addition, and in contrast to prior believes, recent work has shown that, in practice, it is difficult to launch a mimicry attack without being detected [*16*].

A threat to external validity exists in generalizing the results of this study. We have experimented only using four different datasets that focused on Linux kernel. More experiments are required to generalize these results to other operating systems and other programs.

VI. CONCLUSION AND FUTURE WORK

We showed that system call traces when abstracted out in the form of kernel module interactions can result in good detection accuracy while reducing false positive rates. The added value is the low execution time and storage space that kernel module traces have as an advantage over system call events. When put together, we believe that kernel module traces hold real promise in allowing existing anomaly detection such as STIDE and HMM to scale up to large datasets without compromising accuracy. To build on this work, we need to continue experimenting with other datasets. We also need to study malwares and understand situations where attacks do not manifest themselves at the level of kernel module interaction.

REFERENCES

[1] M. Abdel-Azim, A. I. Abdel-Fateh, and M. Awad, "Performance analysis of of artifical neural network intrusion detection systems," in *Intl. Conf. on Electrical and Elctronics Engineering*, Bursa, Turkey, 2009, pp. 385-389.

[2] U. Ahmed and A. Masood, "Host based intrusion detection using RBF neural networks ," in *Int. Conf. on Emerging Technologies*, pp. 48-51, 2009.

[3] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow Anomaly Detection," in *IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2006, pp. 48-62.

[4] S.B. Cho and H.J. Park, "Efficient anomaly detection by modeling privilege flows using hidden Markov model," *Computers and Security*, vol. 22, no. 1, pp. 45-55, Jan. 2003.

[5] G. Creech and J. Hu, "A Semantic Approach to Host-based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1-1, 2013.

[6] G. Creech and J. Hu, "Generation of a new IDS test dataset: Time to retire the KDD collection," in *Wireless Communications and Networking Conf.*, 2013, pp. 4487-4492.

[7] M.C. Desmarais and F. Lemieux, "Clustering and Visualizing Study State Sequences," in *Proc. of 5th Conf. on Educational Data Mining*, Memphis, TN, USA , 2013.

[8] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for Unix processes," in *Proc. of the 1996 IEEE Symp. on Security and Privacy*, Washington, DC, USA, May 1996, pp. 120-128.

[9] A. Gabadinho, G. Ritschard, N. S Müller, and M. Studer, "Analyzing and Visualizing State Sequences in R with TraMineR," *J.of Statistical Software*, 40(4), pp. 1-37, 2011.

[10] A. K. Ghosh, C. Michael, and M. Schatz, "A Real-Time Intrusion Detection System Based on Learning Program Behavior," in *Proc. of the third Intl. Workshop on Recent Advances in Intrusion Detection*, Toulouse, France, Oct.

2000, pp. 93-109.

[11] X. D. Hoang, Jiankun Hu, and P Bertok, "A multi-layer model for anomaly intrusion detection using program sequences of system calls," in *11th Conf. on Network*, 2003, pp. 531-536.

[12] X. D. Hoang, J. Hu, and and P. Bertok., "A program-based anomaly intrusion detection scheme using multiple detection engines and fuzzy inference," *J. Netw. Comput. Appl*, vol. 32, no. 6, pp. 1219-1228, Nov. 2009.

[13] S. A. Hofmeyr, S. Forrest, and and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Security*, vol. 6, no. 3, pp. 151-180, 1998.

[14] J. Hu, X. Yu, D. Qiu, and and H.H. Chen, "A simple and efficient hidden Markov model scheme for host- based anomaly intrusion detection," *IEEE Network*, vol. 23, no. 1, pp. 42-47, Jan. 2009.

[15] G. Jiang, H. Chen, C. Ungureanu, and K.I. Yoshihira, "Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata," in *Proc. 2nd Intl. Conf. on Automatic Comp.*, Seattle, USA, June 2005, pp. 111-122.

[16] H. G. Kayacik and A. Nur Zincir-Heywood, "Mimicry attacks demystified: What can attackers do to evade detection?," in *Proccesdings of , 6th Annual Conf. on Privacy, Security and Trust*, 2008, pp. 213-223.

[17] U. Larson, D. Nilsson, E. Jonsson, and S. Lindskog, "Using System Call Information to Reveal Hidden Attack Manifestations," in *Proc. 1st Intl Workshop on in Security and Communication Networks*, Norway, 2009, pp. 1-8.

[18] P. Lichodzijewski, A. Nur Zincir-Heywood, and M. Heywood, "Host-based intrusion detection using self-organizing maps," in *Proceedings of the 2002 Intl. Conf. on Neural Networks*, Honolulu, USA, 2002, pp. 1714-1719.

[19] A. Liu, X. Jiang, J. Jin, F. Mao, and J. Chen, "Enhancing System Called-Based Intrusion Detection with Protocol Context," in *Fifth Intl. Conf. on Emerging Security Information Systems and Technologies*, pp. 103-108, 2011.

[20] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing*, 7(4), pp. 381-395, 2010.

[21] Metasploit. (2012) Metasploit Pentration Testing Software. [Online]. http://www.metasploit.com

[22] [REMOVED DUE TO BLIND REVIEW].

[23] NVD. (2012) National Vulnerability Database. [Online]. http://nvd.nist.gov

[24] M. Mahoney and P.Chan, "Learning rules for anomaly detection of hostile network traffic," in *Proc. 3rd IEEE Intl. Conf. on Data Mining*, pp. 601-604, 2003.

[25] A. Patcha and J.,M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, 51(12), pp. 3448-3470, 2007.

[26] L.R Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. of IEEE*, 77(2), pp. 257-286, 1989.

[27] W. Lee and S.J. Stolfo, "A framework for constructing features and models for intrusion detection systems.," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 227-261, Nov. 2000.

[28] G. Tandon, "Machine Learning for Host-based Anomaly Detection," Florida Institue of Technology, Melbourne, Florida, USA, Ph.D. thesis 2008.

[29] K. M. C. Tan and R. A. Maxion, "Determining the Operational Limits of an Anomaly-Based Intrusion Detector," *IEEE Journal on Seletected Areas in Communications*, vol. 21, no. 1, pp. 96-110, 2003.

[30] R Development Core Team, "R: A Language and Environment for Statistical Computing," *R Foundation for Statistical Computing*, 2011.

[31] UNM. (1998) University of New Mexico Dataset, http://www.cs.unm.edu/~immsec/systemcalls.htm. [Online]. http://www.cs.unm.edu/~immsec/systemcalls.htm

[32] A. Valdes and K. Skinner, "Adaptive, Model-Based Monitoring for Cyber Attack Detection," in *Proc. of 3rd Intl. Workshop on Recent Advances in Intrusion Detection, LNCS*, France, Oct. 2000, pp. 80-92.

[33] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection system.," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 255-264.

[34] W. Wang, X. H. Guan, and X. L. Zhang, "Modeling program behaviors by hidden Markov models for intrusion detection," in *Proc. of Intl. Conf. on Machine Learning and Cybernetics*, Shanghai, China, Aug. 2004, pp. 2830-2835.

[35] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Proc. of 1999 IEEE Symposium on Security and Privacy*, Oakland, USA, pp. 133-145, 1999.

[36] C. Wohlin et al., *Experimentation in Software Engineering: An Introduction*. Norwell, USA: Kluwer Acad. Pub., 2000.

[37] N. Ye, S. M. Emran, Q. Chen, and S. Vilbert, "Multivariate Statistical Analysis of Audit Trails for Host-Based Intrusion Detection," *IEEE Trans. on Comp.*, 51(7), pp. 810-820, 2002.

[38] D. Y. Yeung and Y. Ding., "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229-243, Jan. 2003.

[39] D. Yuxin, Y. Xuebing, Z. Di, D. Li, and A. Zhanchao, "Feature representation and selection in malicious code detection methods based on static system calls," *Computers & Security*, 30(6-7), pp. 514-524, 2011.