A Host-based Anomaly Detection Approach by Representing System Calls as States of Kernel Modules

[1]Syed Shariyar Murtaza, [1]Wael Khreich, [1]Abdelwahab Hamou-Lhadj, [2]Mario Couture
[1]Software Behaviour Analysis (SBA) Research Lab, Department of Electrical and Computer Engineering,
Concordia University, Montreal, QC, Canada
[2]System of Systems Section, Software Analysis and Robustness Group,
Defence Research and Development Canada (DRDC), Valcartier, QC, Canada
[1]{smurtaza, wkhreich, abdelw}@ece.concordia.ca, [2]mario.couture@drdc-rddc.gc.ca

Abstract—**Despite over two decades of research, high false alarm rates, large trace sizes and high processing times remain among the key issues in host-based anomaly intrusion detection systems. In an attempt to reduce the false alarm rate and processing time while increasing the detection rate, this paper presents a novel anomaly detection technique based on semantic interactions of system calls. The key concept is to represent system calls as states of kernel modules, analyze the state interactions, and identify anomalies by comparing the probabilities of occurrences of states in normal and anomalous traces. In addition, the proposed technique allows a visual understanding of system behaviour, and hence a more informed decision making. We evaluated this technique on Linux based programs of UNM datasets and a new modern Firefox dataset. We created the Firefox dataset on Linux using contemporary test suites and hacking techniques. The results show that our technique yields fewer false alarms and can handle large traces with smaller (or comparable) processing times compared against the existing techniques for the host based anomaly intrusion detection systems.**

Keywords—Host-based Intrusion Detection System, Software Security, Software Reliability.

I. Introduction

The last few years have seen a noticeable increase in the number of intrusions on computer systems across the world, despite recent advances in protective mechanisms and defensive measures. In practice, most host-based intrusion detection systems (HIDS) focus on detecting signatures of known attacks (misuse detection). In an attempt to detect attacks not known previously, host based *anomaly* intrusion detection systems provide a complementary approach. Anomaly detection systems can detect novel attacks by monitoring significant deviations in normal program behaviour. However, these systems suffer from high false alarm rates despite many prior efforts by researcher to reduce them (e.g., [*32*] [*37*] [*34*] [*7*] [*13*]). In addition, the processing time of some of the prior anomaly detection algorithms can be high, such as the time to train the Hidden Markov Model (HMM) on large normal system traces [*34*] [*4*] [*12*] [*13*].

In an attempt to reduce the false alarm rate and processing time, we propose a novel anomaly detection technique based on semantic interactions of system calls of a program. We represent system calls as kernel modules, called states, and analyze the behaviour of the program at the level of interaction of states. We use states to identify anomalies in unknown traces by comparing their proportion of occurrences in normal traces and unknown traces. Analysts can also use this state representation to develop a close rapport with the behaviour of the program and make decisions about anomalies.

In this paper, we focus on Linux-based programs, though our approach is readily adaptable to other operating systems. We evaluated our technique on system call traces collected from four programs (i.e., Login, PS, Xlock and Stide) from the University of New Mexico (UNM) datasets [*7*] [*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 publicly available datasets [*2*] [*5*]. We also evaluated our technique on a new Firefox dataset. We created the Firefox trace dataset by using contemporary software testing and hacking techniques on Linux OS. Each trace in Firefox dataset consists of millions of system calls. It is publicly available from our website: “http://www.ece.concordia.ca/~abdelw/sba/FirefoxDS”.

We compared our technique with well-known contemporary anomaly detection techniques, such as the Sequence Time-Delay Embedding (Stide) [*7*] and HMM [*34*] [*33*]. The results show that the proposed anomaly detection technique is capable of reducing the false alarm rate and the processing time (especially for large traces) compared to Stide and HMM, while maintaining a high detection rate. More importantly, our technique provides the feedback based on state representation that helps in better understanding of the behaviour of a program.

The rest of the paper is organized as follows: Section II presents a brief background and a literature review; Section III describes state representation and anomaly detection using states; Section IV outlines our technique to automatically detect anomalies based on Section III; Section IV presents the case study on datasets; Section VI shows the threats to validity; and Section VII concludes with future work.

II. Background and Literature Review

Traditional Intrusion Detection Systems (IDS) compare software behaviour with a database of known attributes extracted from known attacks. When a pattern of attack is found the behaviour is considered as anomalous, otherwise, it is considered legitimate. These IDS are called misuse or (signature-based) detection systems. Another type of intrusion detection systems works by building a robust baseline of the normal behaviour of a system and then monitors for deviations from this baseline during system operation [*25*]. These systems are called anomaly detection systems and they are our focus.

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 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 [*36*], and using frequency distribution based anomaly detection on shell command logs [*37*]. Another type of HIDS train different algorithms on system calls of normal software behaviour. These systems raise alerts when the deviation from normal system calls is observed in unknown software behaviour (e.g., a trace). Anomaly based HIDS focusing on system calls deviations are related to our work and are described below.

Forrest et al. [*7*] propose to build 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 behaviour and compared with the database. If an unknown sequence is found in a trace then it is considered as anomalous. Hofmeyr et al. [*14*] improve the Sliding Window algorithm by computing the Hamming distance between two sequences to determine how much one sequence differs from another. Warrender et al. [*34*] use a locality frame count (i.e., the number of mismatches during the last ‘n’ calls) instead of the Hamming distance for the Sliding Window algorithm. Warrender et al. [*34*] actually compare the Sliding Window algorithm, Hidden Markov Model (HMM) and Association Rules (called RIPPER) on system call traces. They identify that HMM and Sliding Window are better than the Association Rules, but they did not find any conclusive difference between HMM and Sliding Window.

When Warrender et al. [*34*] 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. [*33*] 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 [*34*]. Similarly, Yeung and Ding [*37*] also employ HMM on system call sequences, and called it dynamic modelling. Yeung and Ding [*37*] 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. [*12*] move a step ahead and propose a multiple layer detection approach in which one layer train the Sliding Window algorithm on system calls and another layer train HMM on system calls. They combine the output of both to detect anomalies. Hoang et al. [*13*] also improve their earlier work [*12*] by combining HMM and the Sliding Window algorithm using fuzzy inference engine. Though HMM has been found useful, one of the main hurdles for HMM is its high training time. Researchers like Hu et al. [*15*] propose an incremental HMM training technique to reduce its training time by 50%.

To detect anomalous system calls researchers have employed standard multilayer perceptron [*1*], Elman recurrent neural network [*9*], self organizing maps neural network [*18*] and radial basis functions based neural networks [*2*]. SVM, decision tree and K-means clustering have also been used on system calls extracted from static analysis [*38*]. The researchers in [*1*] and [*38*] employed training on both normal and anomalous traces to detect anomalies.

Jiang et al. [*16*] extract varied length n-grams from call traces of normal behaviour, and build an automaton that represents the generalized state of the normal behaviour: they use this automaton to detect anomalous behaviour 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 increase 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.

Nonetheless, the fundamental ways of data interpretation employed by all these techniques are similar—manipulation of system call sequences. The research presented in this paper introduces a complete novel way of interpreting the raw system calls by employing a semantic interpretation of system call interactions at the level of kernel modules (states). Though system call sequence analysis has higher sensitivity, the state sequence analysis helps analysts in better system understanding with lesser false alarm rates.

III. Representing System Call Traces As State Sequences

We have developed a technique that transforms system call sequences to state sequences and differentiate state sequences in normal and anomalous traces to detect software anomalies. In Section III.A, we first outline the fundamentals of state sequence representation of system call traces. State sequence representation is used in social sciences to visualize activities [8]. We describe in Section III.B how state sequence representation can be used to discriminate anomalies in software applications. We use TraMineR [8] package in R [30] for visualization.

A. Representation of Temporal Sequences of System Calls

A system call sequence is usually represented as a temporally ordered collection of system calls where each individual system call is an event. An ordered collection of events is called an event sequence. In the event sequence, an event occurs at a specific point of time, has no duration and a transition occurs from one event to another [6]. For example, the system call sequence "open, read, close" is an event sequence. As described in Section II, the host-based anomaly intrusion detection system focus on the use of event sequences (e.g., [34] [33] [28]) to characterize the normal behaviour of software applications and use this as a baseline to detect deviations from normalcy.

Table I GROUPING OF SYSTEM CALLS AS STATES OF KERNEL MODULES

State	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

A temporal sequence of system calls can also be represented as a state sequence. In the state sequence, a state occurs for a specific time duration, and a transition takes place from one state to another [6]. For example, an application can be performing file system operations, memory management operations, network operations, kernel operations and so on. The two concepts of events and states are integrated because state transitions are triggered by events but they differ in their representation and analysis [6]. State sequences are particularly useful in visualization of system call events in traces. For example, if we were to visualize Linux based system call traces by representing every system call with a color then a chart would have resulted in more than 300 (the number of distinct system calls in a typical Linux kernel) colors and become incomprehensible. However, if the system call events are grouped to form valid states then the behaviour of an application can be illustrated on a chart.

In order to project a system call event sequence to a state sequence, we group the system calls by their modules in the Linux source code. Table I shows the list of modules and the total number of system calls that fall in each module[1]. For example, the File System module contains 131 system calls (e.g., open, read, close, etc.). Each module represents one state, and there are a total of eight distinct states in Linux, corresponding to the number of Linux module in a typical distribution. We also divide an event sequence into time units of length ten. A time difference of more than ten units creates a new sequence. We choose length ten because it can accommodate traces of small to large sizes. Event sequences of system calls are then represented as state sequences of kernel modules. Each state sequence shows that an application spends certain amount of time in one state and then moves to another state. An example of state sequences for an anomalous trace of Firefox is shown in Fig. 1.

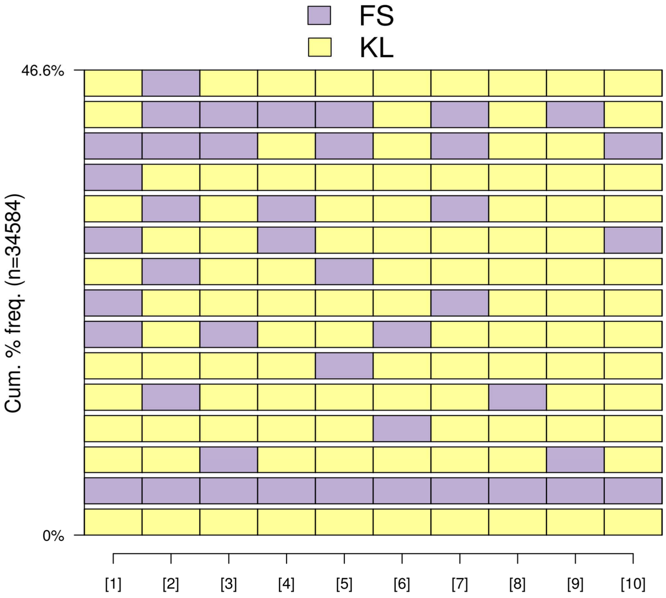

Fig. 1. Fifteen frequent state sequences in an anomalous trace of Firefox constituting 46.6% of 34584 state sequences. (FS stands for File System and KL for Kernel)

Fig. 1 shows fifteen frequent sequences that appear in an anomalous trace of Firefox. First two sequences show that Firefox stays in Kernel state and File System state, respectively. Third state sequence shows that Firefox moves from the Kernel state to File System state twice before moving back to Kernel state. These fifteen state sequences constitute 46.6% of the Firefox trace, showing that most of the time is spent in the Kernel and File System state transitions in this trace.

B. Discriminating Anomalies Through Density plots of State Sequences

Fig. 1 is useful in analyzing individual state sequences that represent a major proportion of a trace. However, the number of sequences can be extremely large, and hence it is important to analyze the whole trace. Density plots can be used to visually summarize occurrences of different states within a trace. Density plots actually represent the proportion of states at each time interval and can be used to measure the probability of state occurrence. Fig. 2 shows the density plots of a normal trace and an anomalous trace of three different subject programs that we use in this study.

[1] The mapping of modules and system calls for Linux can be found at http://syscalls.kernelgrok.com

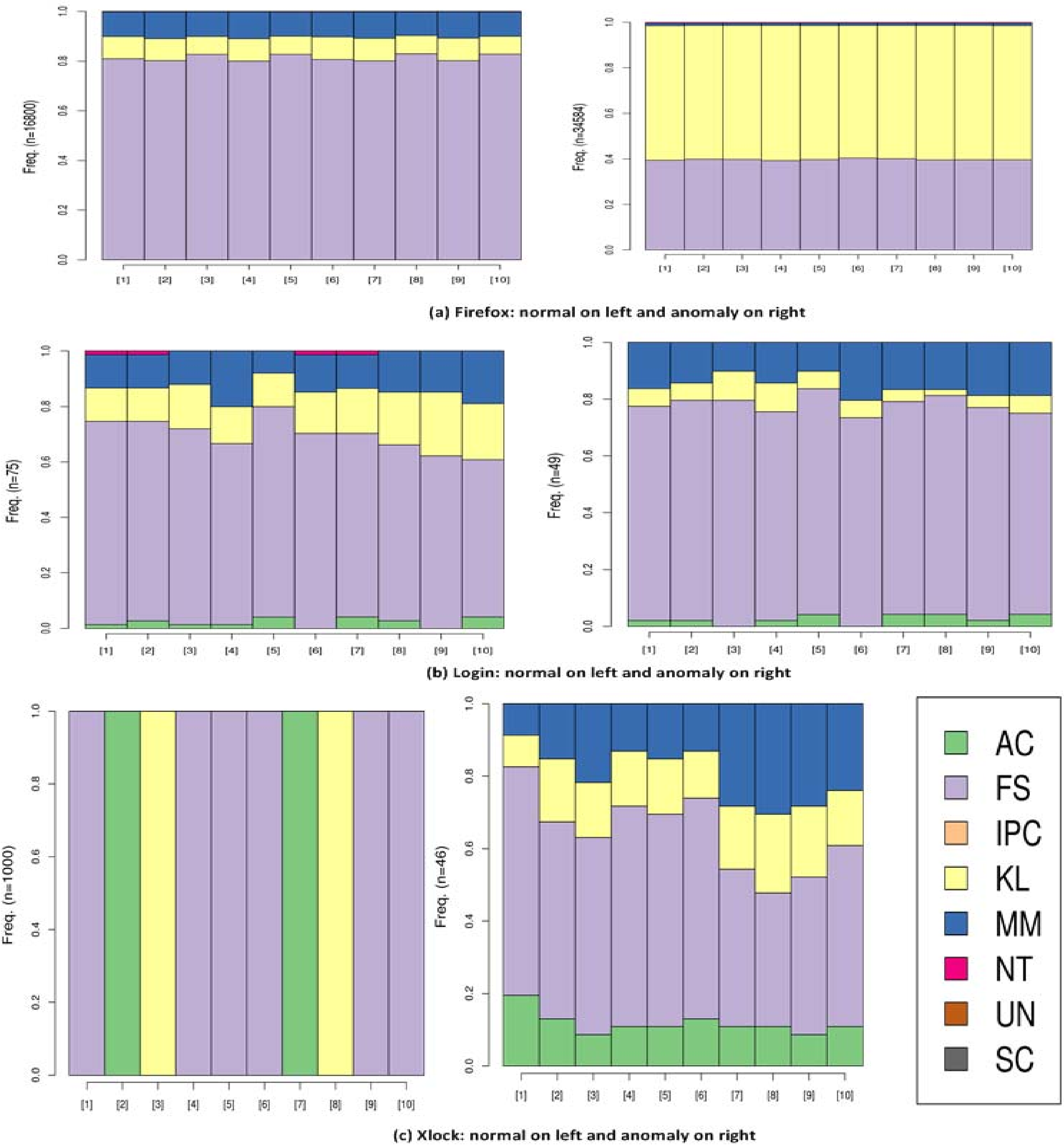

Fig. 2. Density plots of normal and anomalous traces of Firefox, Login and Xlock programs. (Where AC is Architecture, FS for File System, IPC for Inter Process Communication, KL for Kernel, MM for Memory Management, NT for Network, UN for Unkown and SC for Security and ‘n’ shows total number of state sequences in a trace)

Fig. 2a shows that when an attack is launched against Firefox, then the proportion of the Kernel state in the anomalous trace is extremely high compared to the normal execution of Firefox. The attack is actually a denial of service and tries to execute an arbitrary code by exploiting the dangling pointer through tree data structure in Firefox. Similarly, Fig. 2b shows that the Login program produces a higher proportion of the File System state in the anomalous trace than that of the normal trace; whereas, the Kernel, Memory Management and Networking states are lower in proportion to the normal trace. The attack on the Login program is a Trojan attack that allows an intruder to login through backdoor and hide its activities. The difference between normal and anomalous traces is quite subtle in the case of Login program compared to Firefox (Fig. 2a). In Fig. 2c, we show a normal and an anomalous trace of the Xlock utility in Linux. Again the difference in proportions of occurrences of the Architecture, Kernel, File System, and Memory Management states of the two traces are clearly noticeable. The attack in the case of Xlock program is a buffer overflow for one of the command line options.

Due to the space limitation we have only shown one normal trace for each program of Fig. 2. The normal trace shown in Fig. 2 is the representative of the majority of the traces of that program, although some normal traces do have variations. For example, for the Xlock program there are some normal traces which are similar to the anomalous trace shown in Fig. 2c for Xlock. However, similar to the example shown for the Login program, these normal traces for Xlock also have different proportion of occurrences of the File System, Memory Management, Kernel and Architecture states than the anomalous trace. In the case of Firefox, the distinctions between normal and anomalous traces are always clear as shown in Fig. 2.

We have observed that the Kernel (KL), Memory Management (MM), and File system States (FS) are the most commonly occurring states within the traces of considered programs. Deviations in the proportion of occurrences (of these three states) between normal and attack traces may help in detection of anomalies in a program's behaviour. However, the deviations corresponding to an individual state may not be accurate measure of process anomaly. This is illustrated in Table II, where the probability of occurrences of each state (FS, KL and MM) by itself may not discriminate normal from anomalous traces for Xlock.

In Table II, if we compare the probabilities of individual states of normal and anomalous traces then we can see that the probabilities of states in anomalous traces are within the range of normal traces. For example, 0.15 and 0.16 for KL are within the normal values of 0.01 to 0.20 for KL, and so on for FS and MM states. If we compare the FS value of anomalous traces with the closest value (or the closest largest value) then we can see that the difference in the corresponding KL or MM values is much larger than the difference of FS values. For example, the closest FS value to 0.53 in the anomalous trace is 0.54 in normal traces with a difference of 0.01 but the difference in MM and KL is 0.06 and 0.10, which is considerably higher. This difference could well be the regular deviations as in the case of deviations in normal traces when FS is 0.82 and MM is 0.03 and 0.11. However, when we analyze the probabilities of FS, KL and MM in all the normal traces, the maximum deviations observed in KL or MM at 0.54 FS value in normal traces is less than the 0.53 FS value of the anomalous trace. This can also be observed by looking at the closest values of KL/MM and the corresponding FS and MM/KL values. This creates enough doubt to raise an alarm for an anomaly and it is the anomaly.

Table II PROBABILITY OF OCCURRENCES OF CRITICAL STATES FOR XLOCK

Trace #	FS	KL	MM	Type
1	0.60	0.20	0.00	Normal
2	0.54	0.06	0.40	Normal
3	0.73	0.04	0.23	Normal
4	0.74	0.05	0.03	Normal
5	0.82	0.01	0.03	Normal
6	0.82	0.03	0.11	Normal
7	0.55	0.15	0.19	Anomalous
8	0.53	0.16	0.20	Anomalous

Once we identify that a particular trace contains an anomaly, then an analyst can further analyze the individual state sequences (to locate the anomalous ones) by using the frequent state sequence plot as shown in Fig. 1. In fact, Fig. 1 represents the frequent sequences of the attack illustrated in Fig. 2a. The visual illustration of Fig. 1 can help analysts in determining which state sequences are the main causes of an anomaly. In addition, an analyst can look at the density plots to analyze the false alarms. In the next section, we show an algorithm that can be used to detect anomalies automatically using the probabilities of states as shown in the previous example.

IV. TECHNIQUE TO DETECT ANOMALIES

A. Algorithm

Using the observations in Section III.B, we develop a technique, called Kernel States Modeling (KSM), to automatically detect anomalies using the probability of occurrences of states in traces. We partition the normal traces into training, validation, and testing set. We then measure the probabilities of the FS, KL and MM states for each trace in the training set. To ascertain that we have extracted all the possible combinations of FS, KL and MM and that our observation of comparing the maximum differences of states (see Section III.B) will not raise a lot of false alarms, we first evaluate the technique on a validation set. If we find any false alarm for a normal trace in the validation set, we increase the maximum difference threshold by a value of *alpha (alpha* is se*t to 0* initially*)*. We actually iterate through all the traces in the validation set, compare their probabilities of states with the state probabilities of the training set, increment alpha by 0.02 when a false alarm is found, and repeat the whole process on the validation set until there are no false alarms in the validation set. We use this selected *alpha* value to evaluate our technique on the traces in the testing set and on the anomalous traces. The pseudo code for this technique is shown below.

```
Initialize a vector proabilitiesFS
Initialize a vector proabilitiesKL
Initialize a vector proabilitiesMM

/* main function */
function KernelStateModeling (trainFolder,validationFolder,
            normalTestFolder, anomalyFolder)

            trainKSM(trainFolder)
            alpha←validateKSM(validationFolder)
            testKSM(normalTestFolder,alpha)
            testKSM(anomalyFolder, alpha)
 end function
```

```
/* Training */
function trainKSM (trainFolder)
    traceCount<-1
    for traceFile in trainFolder do
        /* Convert a trace of states to a state sequence object*/
        stateSequences ← convertToStateSequence(traceFile)
        /* Get the probabilities of each state */
        probabilityStates ← getProbabilities(stateSequences)
        proabilitiesFS[traceCount] ←probabilityStates["FS"]
        probabilitiesKL[traceCount] ←probabilityStates["KL"]
        probabilitiesMM[traceCount] ←probabilityStates["MM"]
        traceCount<-traceCount+1
    end for
end function

/* Validation */
function validateKSM (validationFolder)
    isAnomaly←FALSE
    for alpha in 0.00 to 0.10 step 0.02 do
        for traceFile in validationFolder do
            isAnomaly←evaluateKSM(traceFile, aplha)
            if isAnomaly == TRUE then
                /* no need to go through all the traces,
                increment alpha and start again for all the traces */
                break
            end if
        end for

        if isAnomaly==FALSE then
            /* stop incrementing alpha because the best alpha is found */
            break
        end if
    end for
    return alpha
end function

/*Testing */
function testKSM (testFolder, alpha)
    isAnomaly←FALSE
    anomalousTraces←0
    totalTraces←0
    for traceFile in testFolder do
        totalTraces←totalTraces+1
        isAnomaly←evaluateKSM(traceFile, aplha)
        if isAnomaly == TRUE then
            anomalousTraces←anomalousTraces+1
            Display density plot
            Display frequent state sequence plot
        end if
    end for
    normalTraces←totalTraces-attackTraces
    Display (normalTraces/totalTraces)*100
    Display (attackTraces/totalTraces)*100
end function

/* Evaluation function used in validation and testing*/
function evaluateKSM (traceFile, alpha)
    /* Convert a trace of states to a state sequence object*/
    stateSequences ← convertToStateSequence(traceFile)
    /* Get the probabilities of each state */
    probabilityStates ← generateProbabilities(stateSequences)
    proabilityFS ←probabilityStates["FS"]
    probabilityKL←probabilityStates["KL"]
    probabilityMM←probabilityStates["MM"]
    isAnomalous←FALSE
    maxFSTrain ← max (proabilitiesFS)

    if proabilityFS >  maxFSTrain then
        sAnomalous←TRUE
    else
        for nearestFS=probabilityFS to maxFSTrain step 0.01 do
            /*Find indexes of all proabilitiesFS values that match
              probabilitiyFS value */
              indexFS← GetIndexes( nearestFS in probabilitiesFS)
            if indexFS NOT NULL then
                /*get the maximum value of probabilitiesKL found
                  at all the indexFS values*/
                maxKL← max( probabiltiesKL[indexFS] )
                /*get the maximum value of probabilitiesMM found
                  at all the indexFS values */
                maxMM←max(probabiltieisMM[indexFS])
                differnceKL← probabilityKL – maxKL
                differnceMM← probabilityMM – maxMM

                if (differnceKL > alpha) || (differenceMM > alpha) then
                    isAnomalous=TRUE
                    /* break because the trace is anomalous */
                    break
                else
                    /* break because the trace is not anomalous */
                    break
                end if
            end if
        end for
    end if
    return isAnomalous
end function
```

We use the tool R [*30*] to implement this algorithm. Prior to the execution of this algorithm, we convert a system call trace into a state sequence trace by replacing a system call with its corresponding state. This algorithm is quite trivial, it is implemented in Java and the mapping of system calls to states is already described in Table I.

B. Evaluation Criteria

We evaluate the accuracy of our proposed 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

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

Equation 2. False positive rate

In the case of FP rate, our equation employs the concept of traces but the traces can have different sizes. We consider that a trace represents a completion of one task by a process or sub-processes of a program. For example, a test case (or a scenario) to test Bookmarks menu in Firefox represents one task. This task is completed by multiple processes of Firefox. The sequence of system calls made by one process of Firefox during the completion of this task forms one trace. This is in conformance with other researchers [*34*] [*37*] who considered all system calls belonging to one process as one trace of the program.

V. CASE STUDY

A. Dataset

In this paper, we use the system call datasets publicly available on the site of University of New Mexico (UNM) [*31*] as a test-bed for our approach. These datasets consist of system call traces belonging to the normal and anomalous behaviour of several programs. The anomalous traces are generated by launching attacks against some vulnerable programs according to public advisories posted on the Internet. UNM datasets are a common benchmark for system call anomaly detection ([*34*] [*14*] [*13*] [*37*] [*33*] [*12*]), and further details can be found in [*31*]. We have selected from UNM datasets the processes that are executed on Linux OS, such as Login, PS, Stide and Xlock. These four programs are summarized in Table III. We have divided the normal traces randomly into three parts: training, validation, and testing as required by our technique (see Section IV.A).

Table III DESCRIPTION OF THE SUBJECT PROGRAMS

Program	**# Normal Traces**			**#Attack Types**	**#Attack Traces**
	Training	Validation	Testing		
Login	4	3	5	1	4
PS	10	4	10	1	15
Stide	400	200	13126	1	105
Xlock	91	30	1610	1	2
Firefox	125 (35)	75 (21)	500 (140)	5	19

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. Each of the four programs includes several traces of one attack (attack type in Table III). The attack executed on the Login and PS program is a backdoor Trojan attack that allows intruder to login through a backdoor and hide its activities. The Xlock program is intruded by a buffer overflow attack through one of its command line options. The attack on the Stide program 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.

Although UNM datasets are still used as one of the main benchmarks for anomaly detection systems, they are more than a decade old and are not representative of current attacks. Moreover, the normal traces in UNM datasets are collected by executing a program for a longer period of time which does not guarantee execution of most of the functionalities of the program. We have created our own system call based dataset for Firefox web browser to address some of these issues.

We have collected traces of normal behaviour for Firefox 3.5 by executing seven different testing frameworks (test suites) [*22*]. Each test framework executes different components and functionalities of Firefox. The test cases cover most of Firefox functionalities to ensure a certain level of completeness of the model of normal behaviour. More precisely, we determine the completeness of normal behaviour of Firefox by measuring its code coverage for test case executions. The execution of seven different test suites result into approximately 60% source code coverage, 5931 passing test case files (corresponding to 43325 test cases), approximately 1.3 TB of traces, and an average of 19,000-400,000 system calls per test case file. Due to such a large set of trace database, we randomly select five different test case files from each test suite for training, three different test case files from each test suite for validation and 20 different test case files from each test suite for testing. Each test case file corresponds to one trace file and each trace is separated into traces of individual processes of Firefox. Table III shows the number of per-process traces for Firefox and the number of test case files in bracket for normal traces.

To collect anomalous traces, we have launched contemporary attacks against Firefox, selected from public advisories [*23*] and public resources, such as MetaSploit [*21*]. We have executed five different attacks on Firefox and collected their corresponding traces (from the beginning to the end of each attack). The first attack is a memory corruption exploit that tries to execute an arbitrary code. Second attack is an integer overflow attack that causes a denial of service and executes an arbitrary code. Third attack exploits dangling pointers in the tree data structure of Firefox causing an arbitrary code execution and denial of service. Fourth attack is a DOM exploit causing memory corruption and the fifth attack is a null pointer exploit causing denial of service. The number of attacks and corresponding traces are shown in Table III.

The traces of Firefox dataset that we use in this paper can be downloaded from our website, “http://www.ece.concordia.ca/~abdelw/sba/FirefoxDS”. The complete dataset of more than 1 TB traces will soon be made public too on our website shortly.

B. Results

We compare our technique, Kernel State Modeling (KSM), with the two most popular algorithms in anomaly detection: Stide (e.g., [*7*] [*14*] [*34*]) and Hidden Markov Model (HMM) (e.g., [*34*] [*4*] [*12*] [*13*]). Stide is popular for its simplicity and efficiency. It is based on the sliding window technique that extracts sequences of length ‘n’ from a trace by moving one system call at a time. 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 compare them against the sequences in an unknown trace. If a new sequence is found in an unknown trace then it is considered as anomalous and normal otherwise. During the experiments, we applied Stide with two different window lengths ‘6’ and ‘10’ – the best performing values found in previous works [*7*] [*14*] [*34*]. In addition, since Stide requires no optimization of parameters (on the validation set), the training is conducted on the traces from both training and validation sets (see Table III).

HMM is a stochastic model for sequential data and hence naturally suitable for modeling events like system call sequences [*26*]. The process determined by a latent Markov chain having a finite number of states, N, and a set of observation probability distributions, each one associated with a state. Starting from an initial state, the process transits from one state to another according to the transition probability distribution. Then, it emits an observation symbol from a finite alphabet (with M distinct observable symbols) according to the output probability distribution of the current state. HMM is typically parameterized by the initial state distribution probabilities, output probabilities, and state transition probabilities.

Baulm-Welch algorithm is used to train the model parameters to fit the sequences of observations [*26*].

We trained several HMMs each with different number of states $(N = \{5,10,15,20\})$ on the training sequences. We then evaluated these HMMs on the validation set to choose the number of states and to determine the decision threshold for testing technique. We evaluated HMM on an unknown trace in the test set by extracting sequences of length 10 and 100 (similar to Stide) and computing their probabilities according to the trained HMM. 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.

Table IV RESULTS ON THE SUBJECT PROGRAMS

Program		**TP rate**	**FP rate**
Login	***KSM (alpha=0.00)***	100%	0.00%
	Stide (win=6)	100%	40.00%
	Stide (win=10)	100%	40.00%
	HMM (states=10)	100%	40.00%
PS	***KSM (alpha=0.02)***	100%	10.00%
	Stide (win=6)	100%	10.00%
	Stide (win=10)	100%	10.00%
	HMM (states=5)	100%	30.00%
Xlock	***KSM (alpha=0.04)***	100%	0.00%
	Stide (win=6)	100%	1.50%
	Stide (win=10)	100%	1.50%
	HMM (states=5)	100%	0.00%
Stide	***KSM (alpha=0.06)***	100%	0.25%
	Stide (win=6)	100%	4.97%
	Stide (win=10)	100%	5.25%
	HMM (states=5)	100%	0.25%
Firefox	***KSM (alpha=0.08)***	100%	0.60%
	Stide (win=6)	100%	44.60%
	Stide (win=10)	100%	49.20%
	HMM (states=5)	100%	1.40%

The recognizing capability of HMM with length 10 sequences remained poor and did not result in the detection of attacks and false alarms in most of the programs. The recognizing capability of HMM in anomaly detection is best when the sequence size is significantly larger than the number of states [*15*], and we found that length 100 sequences as the best indicator of anomaly. Moreover, we selected those states as the best which resulted into smaller processing time, better TP rate and lower FP rate. On Xlock, Stide and Firefox programs HMM training continued indefinitely for more than 10 hours due to large number of sequences ranging from 300 thousand to 35 million. To improve the processing time of HMM on these programs, 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*].

Table IV shows the results for Stide, HMM and KSM. The value of alpha for KSM (obtained using the validation set), windows length for Stide, and the best states for HMM are also shown for comparison. As shown in Table IV, KSM outperforms Stide and HMM in terms of FP rate for most of the programs. HMM and Stide produced varied results for programs. In some cases, their FP rates are quite high and in some cases their FP rates are quite low. This shows that Stide and HMM are more sensitive to the nature of programs than KSM. In the case of Firefox, the FP rate is worst for the Stide. The reason for the worst FP rate on Firefox is that the test suites in Firefox exercised variety of its functionalities and executed different normal system call paths. Stide is extremely sensitive in nature and start to raise anomalies on every new system call sequence. In the case of the UNM dataset, the traces were collected by executing the system for a long time resulting mostly in the execution of similar functionalities of a program and fewer system call variations. This caused fewer false positives for Stide.

Table V EXECUTION TIME INCULDING TRAINING, VALIDATION AND TESTING OF KSM, STIDE AND HMM

	Size of All Traces	**KSM**	**Stide (Python)**	**Stide (C)**	**HMM**
Login	26.2KB	4.46 secs	0.03 secs	0.09secs	56.43 mins
PS	29.6KB	5.14 secs	0.11 secs	0.12 secs	46.24mins
Xlock	47.4MB	1.51mins	12.3mins	1.28mins	13.37 hrs
Stide	36.2MB	5.85mins	8.53mins	4.22mins	2.3 days
Firefox	270.6MB	9.35mins	4.17 hrs	5.01mins	4.03 days

Table V shows the execution time of KSM, Stide and HMM. The execution time shown includes training, validation and testing time for three techniques. The timing information is collected on a machine having Intel core i5, 8GB RAM and 64 bit Ubuntu 12.04. For Stide, we have shown two different timings: one is from our own implementation of Stide in Python and another one is from highly optimized Stide implementation in C available at UNM site [*31*]. Stide is known for its efficiency and our Python implementation of Stide yields smaller processing time on programs with smaller trace sizes but with the increase in size it yielded higher processing time. In the case of KSM, the processing time on the larger traces is much lower than Stide (Python). We also evaluated the time of Stide using publicly available optimized implementation in C [*31*]. The difference between Stide (C) and KSM is only few minutes, considering that R (KSM) tends to be slower than C (Stide). It can be easily said that a commercial implementation of KSM in C could easily reduce (or better) even few minutes between optimized Stide (C) and KSM.

HMM has been notorious for time consumption and unsurprisingly it consumed the largest amount of time on the subject programs from few hours to several days. In practice, traces tend be extraordinary large and complex (see [*10*] [*11*] for a discussion on trace complexity) that parallel computation techniques (e.g., Hadoop) are needed to handle large data. HMM can be applied with parallel computation techniques but it would create an additional overhead on software systems, which is certainly not feasible for client machines.

KSM is efficient in processing time, has low FP rate and provides visual feedback which are the attributes lacking simultaneously in HMM and Stide. Visual analysis is shown in Section III with density plots and frequent state sequence plots. If an alarm is raised by KSM then the analyst can interpret the behaviour of a program by using these plots. Usually, humans are quite powerful in comprehending patterns because of their experiences, and right tools can enhance this capability. The examples that we show in Fig. 1and Fig. 2 are from our experiments on the subject programs described in this section.

Finally, KSM can be executed on multiple processes of the same program simultaneously. That is if a trace contains system calls of multiple processes of the same program KSM would yield similar results; whereas, Stide and HMM yield higher FP rate. For example, in the case of Firefox Stide resulted in 89% FP rate when executed on multiple processes simultaneously and KSM produced only 2% FP rate.

VI. Threats to Validity

We describe threats to validity in four categories: conclusion validity, internal validity, construct validity, and external validity [*35*].

A threat to conclusion validity exists in that some attacks might go undetected because transformation of finer grain events like system calls to higher level states reduces sensitivity of detection. However, all the attacks were detected by KSM on the subject programs. KSM's sensitivity can be increased by decreasing alpha. This might increase FP rate too. We tested KSM by using different training, testing and validation sets and lowering the values of alpha. For example, when we set alpha to little higher than 0 (e.g., 0.02) then we found that FP rate increased to 3.7% in the worst case of Stide. In the case of Firefox FP rate increased to 2% at 0.02 alpha. In other programs, the FP rate remained the same. If alpha is set to 0.02 for all the programs then the results are still better or comparable to the competing algorithms, considering FP rate, efficiency, and visualization. Currently, we select alpha automatically based on the validation set.

A threat to internal validity exists in the implementation of anomaly detection algorithms. We have mitigated this threat by manually verifying the outputs.

A threat to construct validity exists in the use of only FS, KL and MM states for anomaly detection. It is possible that in a program other states may show anomalous behaviour. If another state exhibit anomaly then the proportion of these states will also be affected and the attack will be detected. It is also possible an attacker might create an attack that keeps the proportion of system calls same as normal system. These attacks are difficult to create and can be made further difficult by adding more states (e.g., AC, NT, etc.). This way attacker has to keep track of system calls of many different states which would increase chances of getting detected.

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

VII. Conclusion and Future Work

In this paper, we propose an approach to reduce false alarm rates and processing time by manipulating semantic interactions of system calls. We convert system call sequences to the state sequences of kernel modules to represent the interaction of different kernel modules. We then determine the probabilities of occurrences of states in normal traces. If the probabilities of states in an unknown trace are in similar proportion to the normal traces, then we consider it as normal otherwise we consider it an anomaly. We evaluated this technique on the UNM dataset and a new modern Firefox dataset. We compared our technique with two well-known anomaly detection algorithms Stide and HMM. The results show that our technique results in reduced false positive rate and reduced (or comparable) processing time on large traces compared to Stide and HMM. In future, we plan to extend this work to monitor multiple programs on the operating system to detect attacks in real-time. We also plan to evaluate this technique on different operating systems and on high severity attacks; e.g., rootkits.

Acknowledgment

This research is partly supported by a grant from NSERC, DRDC Valcartier (QC), and Ericsson Canada.

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, ICET 2009*, Islamaabad, Pakistan, 2009, pp. 48-51.

[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] 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.

[7] 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.

[8] A. Gabadinho, G. Ritschard, N. S Müller, and M. Studer, "Analyzing and Visualizing State Sequences in R with TraMineR," *Journal of Statistical Software*, vol. 40, no. 4, pp. 1-37, 2011.

[9] 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.

[10] A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," School of Information Technology and Engineering (SITE), University of Ottawa, Ph.D. Dissertation 2006.

[11] A. Hamou-Lhadj and T. Lethbridge, "Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools," in *Proc. of 10th Conf. on Eng. of Complex Comp. Sys.*, China, 2005, pp. 559-568.

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

[13] 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.

[14] 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, Aug. 1998.

[15] 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.

[16] 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.

[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*, 2011, pp. 103-108.

[20] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381-395, Dec 2010.

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

[22] Mozilla-Testers, , 2012. [Online]. https://developer.mozilla.org/en/Mozilla_automated_testing

[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*, Melbourne, Florida, USA, Nov 2003, pp. 601-604.

[25] A. Patcha and J.,M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448-3470, Aug. 2007.

[26] L.R Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. of IEEE*, vol. 77, no. 2, pp. 257-286, Feb 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.

[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] 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.

[34] 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, May 1999, pp. 133-145.

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

[36] N. Ye, S. M. Emran, Q. Chen, and S. Vilbert, "Multivariate Statistical Analysis of Audit Trails for Host-Based Intrusion Detection," *IEEE Trans. on Computers*, vol. 51, no. 7, pp. 810-820, July 2002.

[37] 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.

[38] 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*, vol. 30, no. 6-7, pp. 514-524, 2011.