11-14-2019

Malware Analysis for Evaluating the Integrity of Mission Critical Devices

Robert Heras

Florida International University, rhera003@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

MALWARE ANALYSIS WITH MACHINE LEARNING FOR EVALUATING THE INTEGRITY OF MISSION CRITICAL DEVICES

A thesis submitted in partial fulfillment of the
requirements for the degree of
MASTERS OF SCIENCE
in
COMPUTER ENGINEERING
by
Robert Heras

2019

To: Dean John Volakis
College of Electrical and Computer Engineering

This thesis, written by Robert Heras, and entitled Malware Analysis with Machine Learning for Evaluating the Integrity of Mission Critical Devices, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

A. Selcuk Uluagac

Kemal Akkaya

Alexander Pons, Major Professor

Date of Defense: November 14, 2019

The thesis of Robert Heras is approved.

Dean John Volakis
College of Electrical and Computer Engineering

Andr´es G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2019

ABSTRACT OF THE THESIS

MALWARE ANALYSIS WITH MACHINE LEARNING FOR EVALUATING

THE INTEGRITY OF MISSION CRITICAL DEVICES

by

Robert Heras

Florida International University, 2019

Miami, Florida

Professor Alexander Pons, Major Professor

The rapid evolution of technology in our society has brought great advantages, but at the same time it has increased cybersecurity threats. At the forefront of these threats is the proliferation of malware from traditional computing platforms to the rapidly expanding Internet-of-things. Our research focuses on the development of a malware detection system that strives for early detection as a means of mitigating the effects of the malware's execution.

The proposed scheme consists of a dual-stage detector providing malware detection for compromised devices in order to mitigate the devices malicious behavior. Furthermore, the framework analyzes task structure features as well as the system calls and memory access patterns made by a process to determine its validity and integrity. The proposed scheme uses all three approaches applying an ensemble technique to detect malware. In our work we evaluate these three malware detection strategies to determine their effectiveness and performance.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

1.1 Motivation

Malware is often attributed as the cause for loss of data, data integrity, and financial damage for a company or institution. Due to the nature of malware and their individual uniqueness, the determination of a compromise or infection can be daunting and difficult as well as challenging to determine exactly how much damage will transpire because of it. There have been many advances in the industry including the rise of the prominent internet-of-things (IoT). IoT includes sensors, actuators, vehicles, home appliances and just about any device that are embedded with electronics in order to interconnect and exchange data. The increasing popularity of these devices makes them inevitable targets for malicious corruption.

With the emergence and rising popularity of IoT and ubiquitous embedded systems it is increasingly common for them to be infected. Many IoT devices are intended to be deployed to serve their function with little to no maintenance and overhead required. Once deployed, many of these devices do not have enough battery or processing power to perform complex operations much less use standard signature or heuristic methods to detect whether or not they have been infected and compromised as it is too inefficient and costly to monitor at the device.

We propose a framework which uses an embedded low overhead agent in each device to extract key feature data as the device is executing its tasks. This approach allows our machine learning algorithms to make a determination as to whether a device has been maliciously compromised by using behavioral analysis on extracted features.

The embedded agents connect to our remote framework in order to transfer the harvested information and our models analyze this information to determine the stability and integrity of each device. The analysis is done remotely and is specific to each device so it is persistent and is dynamic enough that we can detect various types of malware as they are executing even if the malware has attempted to delay its execution. [RTWH11] These IoT devices typically maintain an operating system, such as Android, and our system augments the OS with an agent that collects execution features of running tasks in order to transmit them for analysis. Other operating systems include Windows, Mac, and specific agents per operating system would have to be generated. These agents have an inherent impact delay in application execution as once the agent runs and at a specific rate, it will impact the native application on the device. This process is conducted on an ongoing basis, with the frequency and amount of information transmitted dependent on situational conditions. Once the malicious activity is detected the device can be halted or the task suspended from functioning within the network in order to prevent further execution and the possible spread of the malware. Furthermore, our framework is intended to work with as little overhead and interruption on the end devices. Figure 1.1 depicts an overview of our framework. The framework's models are constructed and maintained from features that are extracted from benign processes as they are executing to provide an understanding of typical executions.

Novel research into new ways of detecting malicious activity is needed and it is needed for a wide range of devices and device types. Researchers have looked into the use of machine learning models in order to create anomaly detectors and malware classifiers. [CLM01] Specifically the use of feature extraction, system call extraction, and the observance of memory access patterns in order to train models that use algorithms such as Support Vector Machines, Random Forests, Logistic Regression, and Naive Bayes.

From a Forensics standpoint, infected devices pose a myriad of difficulties. Devices can have their data altered, erased, or hidden. [UGPL18b] Forensics investigators must now also account for the rise of the prominent internet-of-things (IoT) such as smart fridges and smart home air conditioning (AC) units.

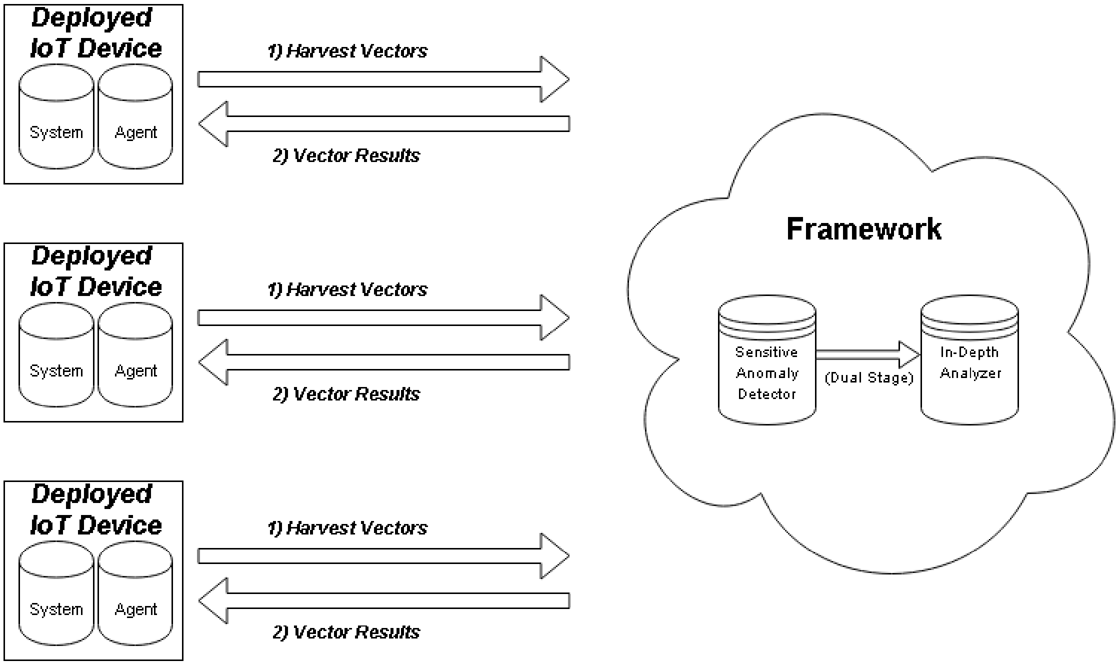


Figure 1.1: Overview of our proposed framework depicting agents, IoT devices, and remote classifier

To combat malware in both home and enterprise environments, we will attempt to use machine learning via anomaly detection and decision tree classifier algorithms to determine if a process is benign or malicious. The frameworks models are constructed and maintained from extracted task structure features, system calls, and memory access patterns harvested from benign and malicious processes as they are executing to provide an understanding of typical executions. The framework will analyze this information to determine the stability and integrity of each device.

Once malicious activity is detected the device can be halted or the task suspended from functioning in order to prevent further execution and the possible spread of the malware. Furthermore, our framework is intended to work with as little overhead and interruption on the end devices.

CHAPTER 2

METHODOLOGY

Our framework functions by analyzing extracted features using a sliding window approach for maximized accuracy. The values of our chosen task structure features, system calls, and memory access patterns are harvested, combined into individual vectors for the current sliding window, and then individually analyzed by our framework. While a malware may obfuscate what system calls it makes, it may not also attempt to cloud what memory access patterns occur or forge its task structure feature values. Using statistical probably built on the results of all three individual models, our framework can accurately determine a processes nature with high true positive rates and low false positives. This ensemble approach offers the maximum coverage of a system and its processes and increases the likelihood of our framework detecting malicious activity.

Our framework uses machine learning to analyze extracted features for the real time detection of malware in deployed devices. We have seen the success of task structure based feature extraction within a Windows [UGPL18a] and Linux environment [UGPL18b]. It has been shown that cloud-based services can provide security to mobile phone devices with limited resources [DA13] and that behavioral based malware detection systems can be used to detect anomalous activity [XZSZ10]. Hybrid approaches for anomaly detection with Hidden Markov Models and naive Bayes models have also been explored and shown to be successful [DA13] [AHHT13] [RTWH11], [XLQZ12].

Our approach differs from other tested approaches as we intend our framework to use an ensemble of features, analyzed via a sliding window in order to be an accurate and dynamic approach, focused on deployed devices. By extracting features from a processes task structure and by observing the system calls [RRL^{+}14] [FHSL96] and the memory access pattern [CCL+09] made, our framework can analyze a devices behavior for anomalies. By observing all three features of a process, our framework offers dynamic detection of malware. Through the use of sliding windows of vectors created from these extracted features, we have increased accuracy for our models. Furthermore, if an anomaly is detected then a more in-depth analysis occurs. Thus far we have performed these extractions on Linux Kernel 4.10, Windows, and intend to further test on Android and various IoT devices. The selection of the Linux operating system as our target is based on the widespread use of Linux dialects for many IoT devices.

Ubiquitous embedded systems or as they're more commonly known, IoT devices, are devices with the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. They have sensors and tend to collect data or act on received information. Such devices are perfect targets for malicious activity. Due to the nature of our framework, a potential application of the basic technology works well for deployed IoT devices. Most IoT devices run command-line interface (CLI) applications and therefore this study aims to build a malware detector for these types of application, as GUI based applications that exist for user interactions are not the norm for IoT devices. IoT devices will also have a minimal number of applications, if not only one. We can focus on verifying one application or a minimal number of applications in an IoT device being monitored. Furthermore, as we know what application(s) are running on an IoT device type, we can develop the anomaly detection models and use them to detect an attack (prior to the device being deployed) since we know the processes running on that specific IoT device. Figure 2.1 depicts the numerous phases of the classification process. Through the frameworks use of drivers, the feature data is first observed and extracted. Then the extracted data is formatted for the model and the classifier performs its classification.

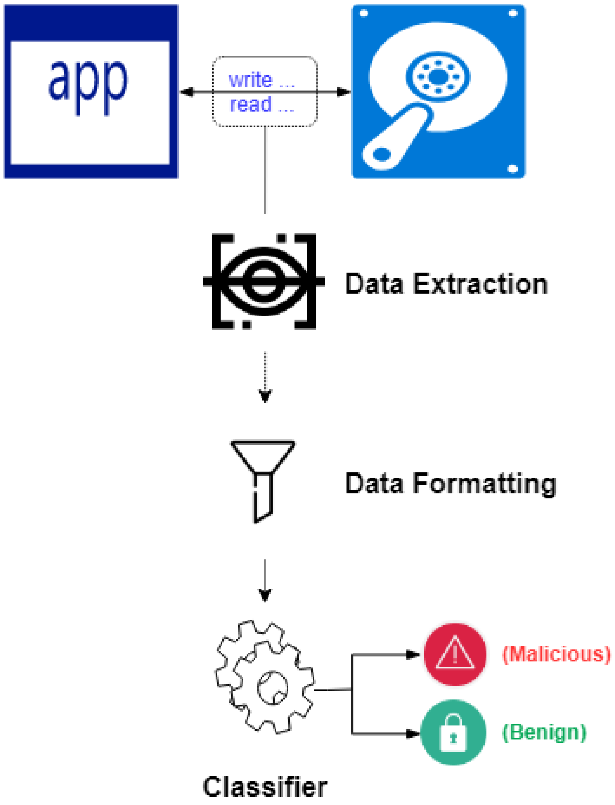
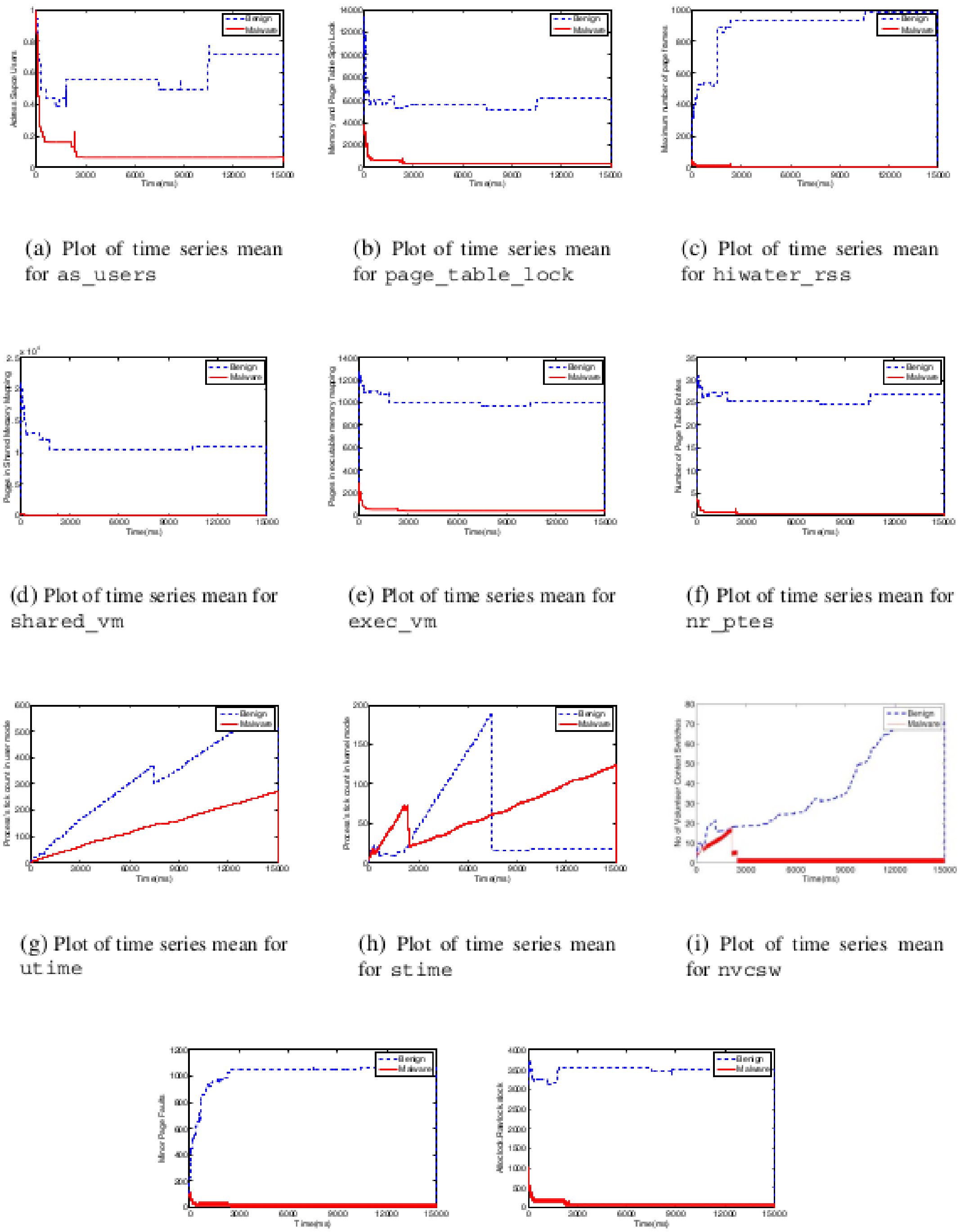


Figure 2.1: A flow diagram depicting the numerous phases of the classification process

2.0.1 Task structure Feature Extraction

For the purposes of malware detection with machine learning via feature extraction, a feature is a system property which should deviate greatly when observed from a benign process versus a malicious process. Each process is manifested in the OS through a data structure called task structure that maintains on an ongoing bases all information associated with the process, like open files, network activity, memory region, time spent in user and kernel space and many more. All of the information that is required to move a process from a running state to a waiting state and at a later time to the running state without affecting the process execution. By observing and cataloging the values of these task structure features, we can map the normal operating procedure the system has and observe any deviations as possibly malicious. The research performed in [SBSF11] determined 11 out of 118 feature which can discriminate between a benign and malicious process with high accuracy, as depicted in Figure 2.2. Within a Linux operating system, a feature logger was used which periodically dumped 118 fields of task structure data every millisecond for 15 seconds. The results of the experiments performed show that a classification decision can be made within the first 100 instances as the process's execution enters a steady state afterwards. The classification power of using task structures for malware analysis is shown as the results were 93% accuracy with minimal overhead, from 50 to 70 microseconds after every millisecond [SBSF11].

For our framework, we opted to extract the eleven identified features every millisecond over the course of fifteen seconds. The eleven features selected showed the most discrepancy between benign and malicious activity out of the identified 118 features. Extractions at a rate less than one millisecond didn't contain enough discrepancy when compared to extractions performed at a rate of one millisecond but caused more overhead due to more extractions over the given 15 second window. Extractions at a rate over one millisecond began to lose their discrepancy and thus resulted in less accuracy. Furthermore, we compared the extracted features of command line interface applications against graphical user interface applications. Due to the event based nature of graphical user interface applications, the models lost accuracy when they were trained on mostly command line interface applications and vice versa. To ensure consistent accuracy, a broad spectrum of applications must be used to train the models.



(a) Plot of time series mean for `as_users`

(b) Plot of time series mean for `page_table_lock`

(c) Plot of time series mean for `hiwater_rss`

(d) Plot of time series mean for `shared_vm`

(e) Plot of time series mean for `exec_vm`

(f) Plot of time series mean for `nr_ptes`

(g) Plot of time series mean for `utime`

(h) Plot of time series mean for `stime`

(i) Plot of time series mean for `nvcsw`

(j) Plot of time series mean for `min_flt`

(k) Plot of time series mean for `slock`

Figure 2.2: Eleven identified task structure features with the greatest disparity when Benign vs. Malicious [SBSF11]

Of the extracted features, one to note is map_count which is an integer value depicting the number of memory regions. Figure 2.3 clearly denoted the disparity between the map_count value of a benign and malicious program execution. The feature hiwater_rss is the high-water of resident set size usage which is used to show how much memory is allocated to that process and is in random access memory. The utime feature depicts how much time a process spends in user space where as the stime feature represents how much time a process spends in kernel space. Each feature in a processes task structure has a purpose and value which can denote its maliciousness. In order to achieve our observations depicted in Figure 2.3, we observed the task structure of various benign processes and various malicious processes and harvested all eleven features. For each benign and malicious process, we graphed the average map_count value at each interval of extraction resulting in Figure 2.3; denoting that the map_count value for a benign process deviates from the standard map_count value for a malicious process. It is worth noting that our other ten extracted features behaved similarly and support the research performed in [SBSF11].

We determined 11 out of 118 task structure feature which can discriminate between a benign and malicious process with high accuracy [SBSF11]. The features we have identified are extracted using a device driver or through introspection [UGPL18b] [SBSF11] and used to train and test our model. These features are then used to audit the running system and perform monitoring on system performance [ZZSL15]. Table 2.1 depicts twelve possible features that can be extracted of the identified 118 in [SBSF11]. Utime for instance is the amount of time that a process spends in userspace. Conversely, stime is the amount of time a process spend in kernelspace. Map_count is an integer value depicting the number of memory regions used by the process. Each feature serves a specific purpose within the task structure

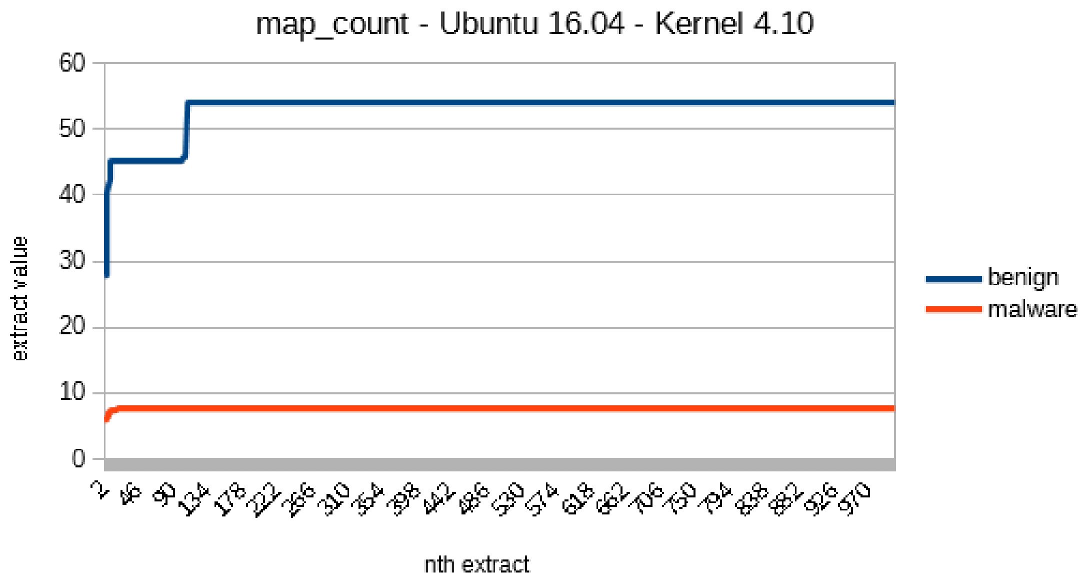


Figure 2.3: Time series statistics of the map_count task structure feature of a Benign vs. Malicious process

of a process and the features value can denote whether that process is benign or malicious. The observations of such features offers a basis for the behavioral analysis of a deployed device.

Table 2.1: Twelve possible Task Structure Features as identified by [SBSF11]

exec_vm	fscount	hiwater_rss	hiwater_vm
map_count	min_flt	nlvcsm	nr_ptes
nvcsm	stime	total_vm	utime

In [ZZSL15], a Feature Extraction and Selection Tool (FEST) was developed which used a feature-based machine learning approach for malware detection and tested on Android platforms. The researchers developed a means to look at the permissions, API actions, IP, and URLs within an android apk and using statistical analysis, collect the features they wish to use. The data-set contained 7972 apps with 50% being malware and the rest being benign. From their research, they discovered that the performance of the Naive Bayes (NB) algorithm is the worst of all algorithms because of its strictly independence limitation and continued further testing with the K-Nearest Neighbor (KNN) and the Support Vector Machines (SVM) algorithms which displayed increased accuracy.

Researchers have also compared the performance of Naive Bayes (NB), Random Forests (RF), and Support Vector Machines(SVM) and determined that the Random Forest algorithm had better classification of malware [JS17]. This was tested with 10-fold cross validation over a sample set of 25 benign and 84 malicious samples. The detection rate for their approach was calculated using five metrics, True Positive Rate (TPR), False Positive Rate (FPR), Precision, Recall, and Accuracy. For static analysis, the Random Forest classifiers had an accuracy of 69.72% and was closely followed by Naive Bayes classifiers with an accuracy of 68.23%. With dynamic analysis, the Random Forest classifiers provided 63.3% accuracy and while the closest other classifier was the Support Vector Machines (SVM) with 60.55% accuracy. Overall, SVM was showed to have a better accuracy than Naive Bayes and Random Forest had the most accuracy among all the classifiers tested.

Evasion of feature extraction based detection might seem possible, however can be quite difficult. [SBSF11] found that the accuracy of their model is unacceptable once eight features are forged, however each parameter depends on a particular configuration of the system - cache, RAM, paging, etc. The values can change from one host to another so a malicious actor must first estimate the values for a particular system and to do so would require the execution of a malware which hooks into sample fields for benign processes. In Linux, this is only allowed to super user processes and as such, a malicious process would be unable to easily evade a feature extraction based malware analysis.

2.0.2 System Call Extraction

Instead of using feature extraction for task structure, researchers have focused on analyzing system calls as an alternative. System calls allow user level applications to interact with the kernel and benign processes interact with the kernel differently than malicious ones. By observing the system calls a process makes, we can determine if it is operating within normal functionality or if it acting in a malicious manner. Such research as in [CMK15b] compares the accuracy of n-gram analysis to that of bag-of-n-grams as well as ordered and unordered 2-gram representations of the system call trace.

The conclusion of the [CMK15b] showed that the best extraction techniques used a minimum system call trace length of 1,000 ordered 3-grams chosen using recursive feature elimination (RFE). Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest features until the specified number of features are reached. The algorithms with the highest accuracy were Logistic Regression (LR) and Support Vector Machine (SVM) which outperformed signature-based detectors. For classification, the Random Forest (RF), K-Nearest Neighbor (KNN), and Logistic Regression (LR) algorithms provided the highest accuracy, outperforming Naive Bayes (NB) and Nearest Centroid Classifier (NCC).

Along side task structure features, we observe the system calls executed by a process. System call extraction has been explored on various platforms such as Hadoop [LHC11], QEMU [BSM13], Android [XLQZ12] [JMJ18], and within Linux using a Loadable Kernel Module (LKM) [SN08] to protect against kernel level rootkits [DGSTG09]. Our framework uses a form of introspection, similar to the extraction of task structure features, in order to extract the system calls made by a process. After testing, we have opted to format our extracted system calls into 3-gram chunks instead of using an approach like bag-o-words for a higher accuracy within our sliding window [CMK15a] [CMK15b] [FT17]. Figure 2.4 depicts the fundamental difference between 1-gram (unigrams), 2-gram (bigrams), and 3-gram (trigrams).

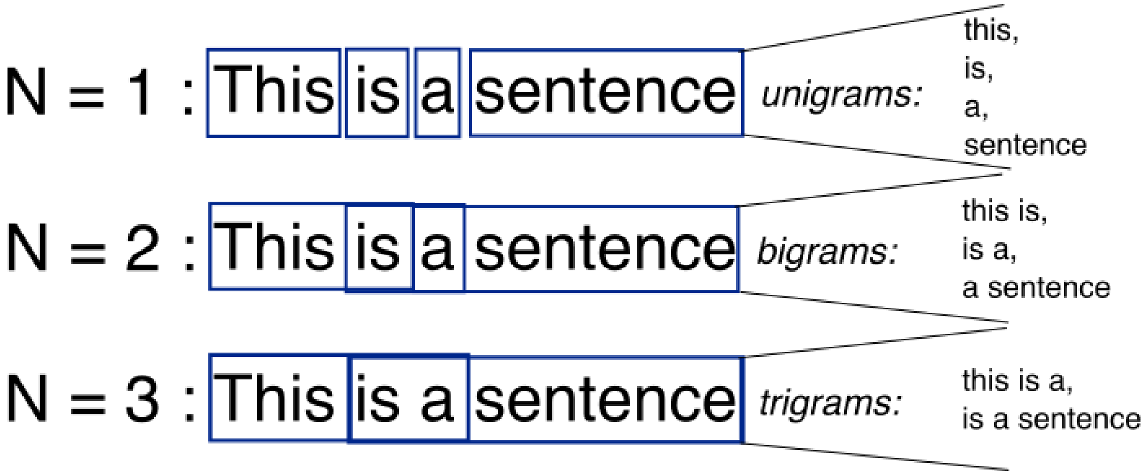
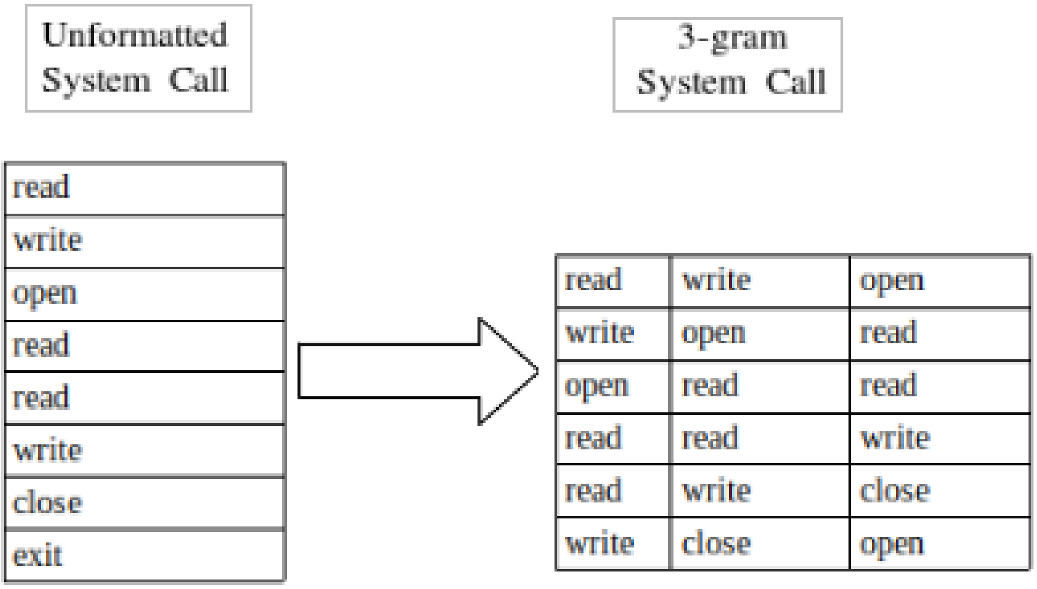


Figure 2.4: Depiction of the fundamental differences in n-gram sizes

The program we use to collect system calls is strace which is a diagnostic and debugging utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel. The strace output is a list of system calls made by a running process for a given duration. Each line in the trace contains the system call name, followed by its arguments in parentheses and its return value. For example ”open(”/dev/null”, O_RDONLY) = 3” is one of the outputs of the open system call when I ran the command ”cat /dev/null”. For our purposes, we need the system call made and not the parameters or the return value. This is list of calls is then formatted into 3-grams. Figure 2.5 depicts the 3-gram formatting of extracted system calls into vectors that are used by the machine learning models. These overlapping system calls provide a depiction of the systems running state and by observing them in order, we can determine behavioral patterns.

Within [CMK15a], more exploration was done including Naive Bayes, SVMs, decision trees, and neural networks. System calls extracted and compiled into 3-gram with a trace length of 1500 calls and compared against a LR, RF, and SVM classi-



Unformatted System Call
read
write
open
read
read
write
close
exit

3-gram System Call		
read	write	open
write	open	read
open	read	read
read	read	write
read	write	close
write	close	open

Figure 2.5: Depiction of 3-gram Formatting of extracted System Calls

fier. According to the results, the worst performing classifiers were Naive Bayes and nearest centroid conversely nearest neighbor and random forest classifiers providing nearly identical performance. The LR classifier performed similarly to the nearest neighbor and the random forest classifier, however the LR performance degraded significantly when the feature reduction techniques singular value decomposition (SVD) and linear discriminant analysis (LDA) were used. Classifier accuracy was shown to improve when n was increased, in particular there was a large increase in performance when moving from 1-gram to 2-gram indicating that the frequency information about system calls alone is insufficient for classification. At a trace length of 1500, the average execution time of the studied processes was 205 ms. This result indicates that high classification accuracy was achievable during the initial execution of the malware samples. As noted by the author, for a production deployment of a classification system, the LR classifier is preferred to the random forest and nearest neighbor.

Research into system calls is an ever expanding niche. [CMK15b] tested bag-of-2-gram representations of system call traces with the use of sliding windows. The ordered 2-gram representation considers the local ordering of the calls within the sliding window, whereas the unordered 2-gram representation ignores the local ordering. For n size 2, 3, and 4 the LR and SVM detectors offered significantly improved detection performance while LR and SVM detectors peaked around 1000 system calls for n sizes of three and four indicating that the studied malware sampled were detected during the early stages of their execution. The research further shows the results of testing a LR detector trained with 10-fold cross validation and n-gram sizes of three, containing 3,500 ordered system calls. Figure 2.6 shows the maximum true positive rate (TPR) as varying sequence lengths.

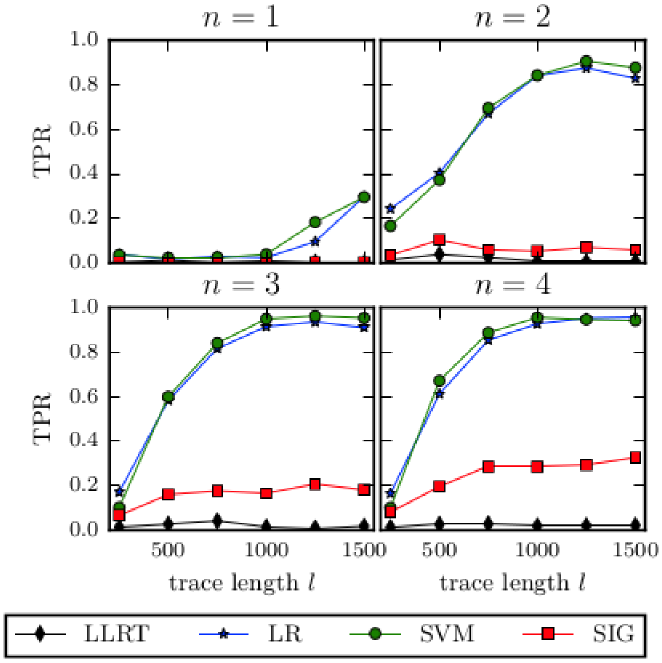


Figure 2.6: Maximum True Positve Rate and False Positive Rate at a sequence length of {1, 2, 3, 4} and system call trace lengths [250, 1500][CMK15b]

Malware analysis on Windows and Linux has been explored however in [LHC11] the Hadoop platform was tested. Using a system calls analysis method with MapReduce, the Hadoop platform is used to analyze the system calls on a server rather than on the clients side. The author also proposes a more universal and persistent model to correlate system calls among modules. Some limitations with this research is if a persistent malware's behavior is completed without the use of system calls then the approach fails to detect the malware. Another limitation is that the cost of data transmission required to have the server analyze the collected system calls has not been measured. The research demonstrated the ability to improve upon previous research which resulted in a 28% increase in detection rate.

Within [BSM13], we see the use of QEMU-emulated sandbox's virtual hard drive where the program Procmon was responsible for collecting system call information for all processes. Once collected, the system calls were analyzed and compared two at a time in a sliding window format. Specifically, the research used Jaccard distance due to good results and clear semantics. However, some limitations encountered was with the restrictions of a virtualized sandbox. Because some of the malware samples were unable to connect to the internet, some of their behavior may have gone unseen and at times malware may even terminate early resulting in insufficient traces which could lead to poorer results.

In [SN08], the researchers develop a lightweight monitoring infrastructure which extracts runtime information from the Linux kernel for the purpose of enhancing system reliability. The loadable kernel module titled KOMI observes the guest OS and intercepts system calls' to collect information from the kernel and suggests strategies to audit and protect the kernel with minimal CPU overhead. The researchers found the monitoring service easy to customize with minimal penalty to

Table 2.2: [SN08] System Call Interception Overhead from KOMI

System Calls	Linux (microsec)	KOMI (microsec)
null	4.0172	4.0173
read	4.4833	4.4833
write	4.2613	4.2615

performance allowing for easy integration into existing embedded systems. The interception overhead between Linux and KOMI is depicted in Table 2.2.

In [ZLK$^+$07], a Loadable Kernel Module (LKM) is proposed as part of a secure auditing system in which system calls are intercepted in order to suggest strategies to audit and protect the kernel. Within [DGSTG09], a system is proposed with the intention of protecting the OS kernel from rootkits which hide their respective running processes and threads. This is done by generating signatures for the kernel's data structure and building a profile using this observed data. Twenty applications were tested and 221 fields were observed, 32 of which were never accessed during the execution of the profiled applications. The research encourages a systematic way of determining which features should be used when determining a data structure for research which extends to machine learning based applications.

With the expansion of embedded systems, smartphones are at risk as well. In [JMJ18], dynamic analysis was performed on an android device using system call extraction. System calls for twenty normal applications and forty malicious applications were recorded for thirty seconds and then sixty seconds while the total amount of times in which the system calls were made was observed and recorded. For example, the epoll_wait operation was called 2613 times by a benign application conversely to 4703 times from a malicious application. Similarly, clock_gettime was called 3796 times by a benign application and 17,251 times by a malicious one. Of note, malicious applications also request high-level permission from users in order to compromise the system by exploiting the granted permission. The research shows that observed system calls made by malicious applications can be used to create a signature to detect malware and although this research does not include machine learning components, it proved that the analyzing of system calls can aid in determining whether a system has been exploited.

In a similar research, [XLQZ12], a Hidden Markov Model was used with system call extraction and key press patterns in order to attempt malware detection on a smartphone; which showed promise. Specifically, 108 system calls were obtained and graphed as shown in Figure 2.7 where y_1=zp is the benign process and y_2=yp is the malicious process, showing that system calls can be used to accurately classify a process using a Hidden Markov Model (HMM).

2.0.3 Memory Access Pattern

While feature and system call extractions have been shown to accurately detect whether a process is benign or malicious, other researchers have begun to explore the same functionality using memory access patterns. Memory access patterns are the patterns in which a system reads and writes data to secondary storage and random access memory. Our framework focuses specifically on virtual memory access rather than physical memory access. By observing what virtual memory access is done by a process and analyzing what portions of virtual memory are interacted with, we can determine if a process has been tampered with. The collected data is compiled, similar to the 3-gram approach of our system calls, and then passed to our first stage detector.

An epoch is the date and time relative to which a computer's clock and timestamp values are determined. Virtual memory is divided into memory regions based

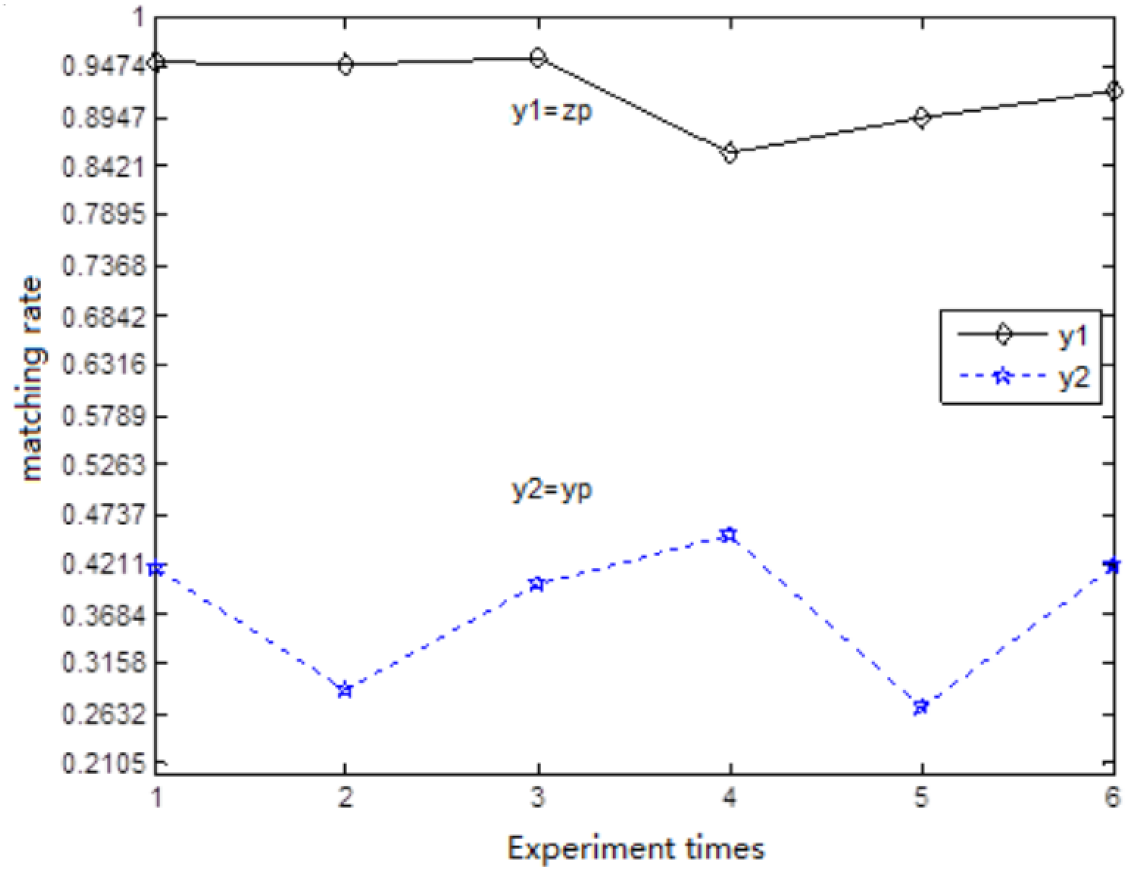


Figure 2.7: Results of Malware detection using a Hidden Markov Model [XLQZ12]

on the systems epoch which are used to build the systems memory model according to the type and frequency of access within these memory regions. Operating systems use pages which are the smallest fixed-length contiguous blocks of virtual memory possible. These pages are allocated together into a page frame which is the smallest fixed-length block of physical memory to which the pages are mapped to. [XRSM17]

In this case, memory addresses are observed for changes that deviate from the norm in order to classify the process invoking the memory access. In [XRSM17], a novel framework is developed which includes techniques for collecting and summarizing memory access patterns. The framework also functions on a two-level

Table 2.3: Details of the performance of our memory access patterns model

	Precision	Recall	F1-score	Support
Benign	0.40	0.02	0.03	93
Malware	0.89	1.0	0.94	777
Average/Total	0.84	0.89	0.85	866

classification architecture and observes kernel rootkits vs user-level malware and contains epoch-based monitoring. Thus far, ten rootkits have been observed using QEMU with the machine learning algorithms having been trained on four of those rootkits and asked to detect six rootkits it has never seen before. The two algorithms tested were Random Forest and Logistic Regression and both showed great accuracy and demonstrated the framework could detect new malware.

For our purposes, we use a program called Perf to observe memory access. The output contains references such as “LFB hit” or L1 or L2 hit” or “Local RAM hit”. These ”hits” are memory access events that occur for both read and write events. Observing their order and occurrence gives us a standard for normal behavior when observing benign processes. We also observe whether these memory access patterns occur in userspace vs. kernel space. Each type of ”hit” is given a correlating numerical value (”LFB hit” = 1, ”L1 or L2 hit” = 2, etc) and this formatted output is used to train our models. Because of the limited number of occurrence types that exist (”LFB hit”, ”L1 or L2 hit”, etc), it is viable to observe these for behavioral analysis.

2.0.4 Data Extraction Process

For task structure feature extraction, we used a loadable kernel module which hooked into the kernel and intercepted task structure features when a process was executing. These features were stored in a buffer as the process was observed in its running state and then the buffer was saved to a file. The resulting feature values were stored as a comma-delimited file and compiled together to train our task structure models. Because each process that is observed creates it's own output file, there is a resulting abundance of data files. As part of our framework we also developed a program to compile all the extracted task structure feature files into one file which could be used to train our machine learning models. Table 2.4 depicts six of the eleven task structure features we extracted using our loadable kernel module. The rows are a sampling of the values these features can have which create a vector that is then analyzed by our models, $\{F_1, F_2, \ldots, F_n\}$. Table 2.4 denotes the first three sampled vectors of a benign process while Table 2.5 denotes the first three sampled vectors of a malicious process. A comparison of both tables depicts the disparity between a benign and malicious process.

Table 2.4: First three task structure feature values from a benign process as extracted using our loadable kernel module

Class	map_count	hiwater_rss	hiwater_vm	total_vm	exec_vm	nr_ptes
Benign	1	0	0	33	0	1
Benign	8	1	1052	546	503	5
Benign	16	1	1865	1865	770	8

Table 2.5: First three task structure feature values from a malicious process as extracted using our loadable kernel module

Class	map_count	hiwater_rss	hiwater_vm	total_vm	exec_vm	nr_ptes
Malware	37	188	3948	6484	1251	16
Malware	42	192	4143	6484	1256	16
Malware	48	202	4258	6484	1261	32

In order to extract the system calls performed by a process we used a program called strace. Strace is a diagnostic and debugging userspace utility for Linux which is used to monitor and tamper with interactions between processes and the kernel via system calls. Strace is a intercepts and records the system calls made by a process and the signals which are received as a result. The name of the system call, its arguments, and its return value are all printed to data files. The extracted system calls must be parsed and our framework does this by iterating through each line in the data files and extracting each call. The calls are then compiled into a vector of three calls per vector as depicted in Figure 2.5.

The memory access patterns of the benign cli application and malicious executable listed in Table 2.3, are obtained using an application called Perf. This tool is a performance analyzing utility within Linux which provides a number of subcommands that allow statistical profiling of the entire system. The commands used in conjunction with this tool recorded both read and write memory access events and outputted them to text files for further processing/formatting. These memory accesses were captured with their parameters and were outputted one per line. Table 2.6 depicts three lines of extracted memory access patterns though we have only included seven of the thirteen available data columns that are outputted by Perf. The symbol column describes if the memory hit is occurring in userspace as depicted by ”[.]” or in kernelspace as depicted by ”[k]”. The object column can inform us if a memory access event is occurring in the heap or the stack. The last column, ”TLB access” describes the unique hits that occur to the translation lookaside buffer (TLB) which is a memory cache that is used to reduce the time taken to access a memory location.

Table 2.6: Example extracted memory access patterns for a benign application

Access Type	Overhead	Samples	Local Weight	Memory Access	Symbol	Object	TLB access
read	1.41%	1	2516	LFB hit	[.]	anon	L1 or L2 hit
read	1.37%	1	2444	LFB hit	[.]	[heap]	L2 or L3 hit
write	0.53%	1	940	Local RAM hit	[k]	[kernel.kallsyms]	L1 or L2 hit

Our framework formats and compiles all the memory accesses into readable data for our model. The formatter within our framework iterates through each line of the file created from the extraction phase and first adds necessary spacing in order to distinguish between features/columns. It then formats each column in the file by representing each possible value for each parameter by a number ranging from 1 to n, with n being the total number of possible values that parameter can take. Once each value has been replaced with a corresponding number, the script continues to iterate through the remaining lines in the file and performs the same actions on the data. The formatted file is then input to the machine learning models to make predictions as to whether or not a specified process is malicious. It operates by creating numerous decision trees during training.

2.0.5 Machine Learning Models

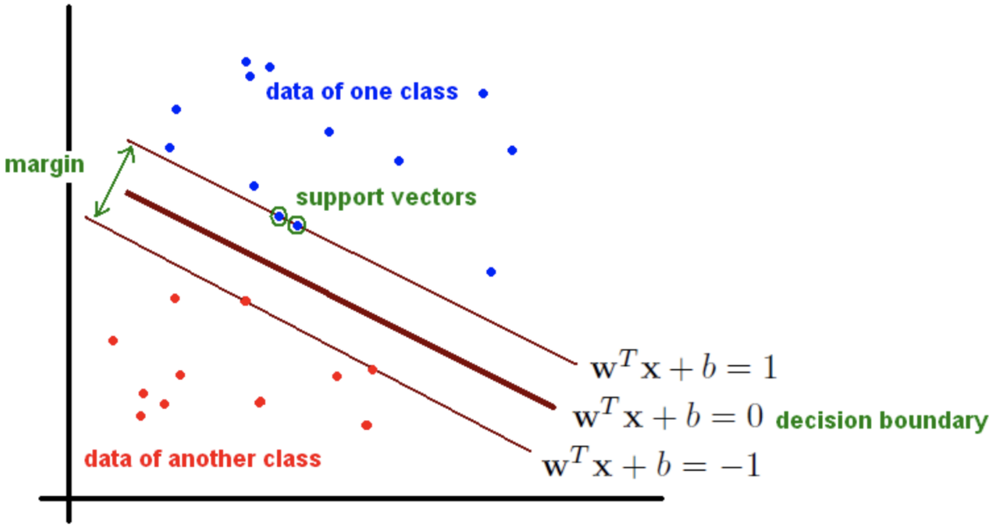

data of one class
margin
support vectors
$\mathbf{w}^T\mathbf{x} + b = 1$
$\mathbf{w}^T\mathbf{x} + b = 0$
decision boundary
$\mathbf{w}^T\mathbf{x} + b = -1$
data of another class

Figure 2.8: Two dimensional depiction of a Support Vector Machine (SVM) algorithm

A Support Vector Machine (SVM) is a discriminating classifier which separates occurrences into two spaces. In two dimensional space, this can be described as a line dividing a plane in two parts where each class lays on either side of the line. Figure 2.8 is a two dimensional depiction of a Support Vector Machine algorithm, it's margins, and it's decision boundaries, as well as where class data and outliers occur. Support vector machines are supervised learning models that can be used to analyze data, recognize patterns, and can be used for both classification and regression tasks. This model is particularly useful where there is much benign data and not much anomalous activity that you are trying to detect, often the case for malware. A fine example of when to use a support vector machine is to detect fraudulent transactions where you might not have many examples of fraud but you can train your classification model on an abundance of good transactions. This is useful for anomaly detection because the scarcity of training examples are exactly what define an anomaly: that is, something that deviates from what is standard, normal, or expected.

A Random Forest algorithm is another type of supervised learning classifier which creates a forest of decisions by some way and randomizes it. There is also a direct correlation between the number of trees and the classifiers accuracy with a larger number of trees resulting in a more accurate model. A few advantage of using the Random Forest algorithm are that the classifier won't overfit the model, the algorithm can handle missing values well, it can be modeled for categorical values, and it can be used for both classification and regression tasks. Figure 2.9 depicts the decision tree classifier of a Random Forest algorithm.

Logistic Regression is a technique originating from the field of statistics. It is a classification algorithm used to assign observations to a discrete set of classes. The Logistic Regressions algorithm transforms its output using a sigmoid function and

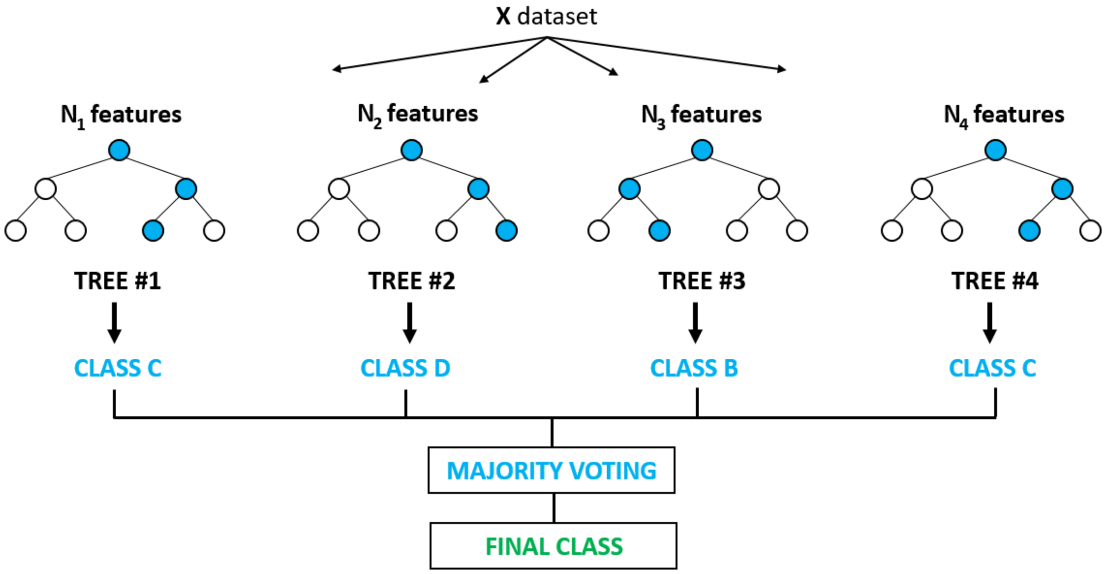


Figure 2.9: Depiction of a decision tree classifier of a Random Forest algorithm

returns the probability value which can be mapped to two discrete classes. Often it is a go-to method for problems with only two classes such as benign and malicious, also known as a binary classification problem. The algorithm is depicted as an S-shaped curve, otherwise known as the boundary, where values are mapped to either side of the curve. Figure 2.10 depicts a Logistic Regression algorithm, it's boundary, as well as true and false samples.

K-fold cross-validation is a statistical method used to estimate the skill of machine learning models. When applied to models it often results in estimates are generally less biases than other methods. The original sample is randomly partitioned into k equal sizes of sub-samples, in our case ten. Of these ten sub-samples, a single sample is kept as the validation data for testing the model while the remaining nine samples are used as training data. The cross-validation process is then repeated ten times with each of the ten sub-samples used exactly once as validation data. The ten results are then averaged to produce a single estimation of the models

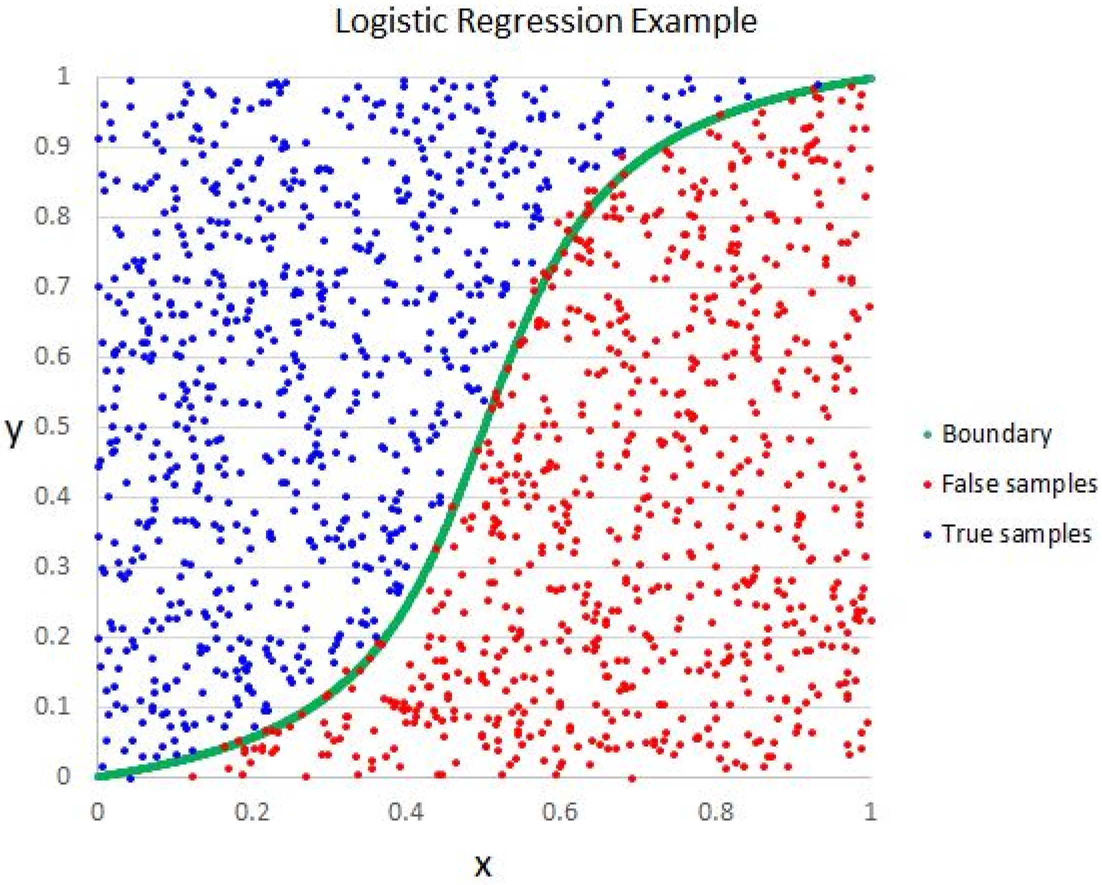


Figure 2.10: Depiction of Logistic Regression algorithm including the boundary and true/false samples

accuracy. The advantage of this technique over repeated random sampling is that all the samples are used for both training and validation and each sample is used only once for validation. Furthermore, in repeated cross-validation the data is randomly split into k folds several times. The performance of the model can therefore be averages over several runs. The goal is also to test a models ability to predict new data that was not used in estimating it in order to indicate problems such as over-fitting or selection bias and to give insight into how the model will independently generalize a dataset. Due to the aforementioned details of cross-validation, we used 10-fold cross-validation as our validation technique for our Random Forest

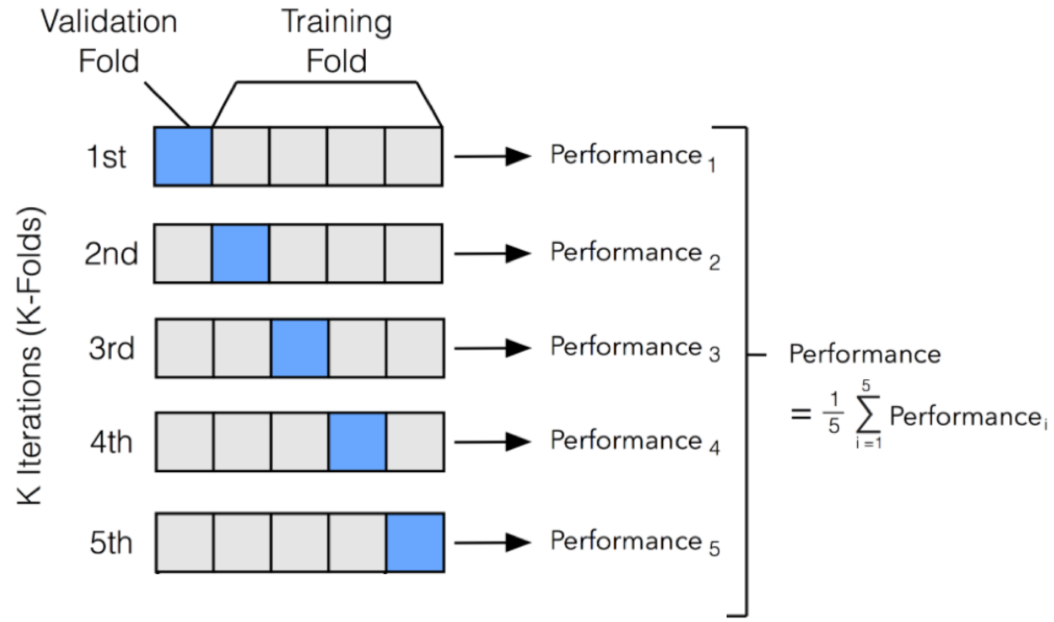


Figure 2.11: Depiction k-fold cross validation and training for machine learning models

and Logistic Regression models. Figure 2.11 depicts k-fold cross-validation as it is used to validate machine learning models such as Random Forest and Logistic Regression.

2.0.6 Ensemble Comparison

Our framework uses feature extraction of task structure properties, system calls, and memory access patterns in order to facilitate a multi-pronged approach to malware detection and analysis. The task structure of a process describes the shared resources used such as an address space or open files. The kernel stores the list of processes in a linked list and each process contains a task structure which contains all the information about that specific process. System calls are the fundamental interface between an application and the Linux kernel and are invoked to perform an action.

Some common system calls are read, write, and open. A memory access pattern is the pattern with which a system or program reads and writes memory to secondary storage.

We have identified various features within a Linux environment whose values deviate sufficiently over time when they are benign or malicious. These features consist of the smallest subset possible to provide detection of malware [SBSF11] [CMK15b] [XRSM17]. These features were selected after much testing and observation to determine their deviance in value when they were benign or malicious. Thus far we have performed these extractions on Linux Kernel 4.10, and intend to further test on Windows, Android, and various IoT devices. The selection of the Linux operating system as our target is based on the widespread use of Linux dialects for many IoT devices.

Our task structure, system call, and memory access features are harvested individually and across different system times. A vector is a collection of extracted features of a given feature type (task structure, system calls, memory access). Each vector has a given formatting requirement given which feature type it belongs to. For instance, task structure and memory access vectors are a collection of extracted features $\{\mathrm{F}_1, \mathrm{F}_2, \ldots, \mathrm{F}_\mathrm{n}\}$ while system calls are 3-gram representations of extracted system call features $\{\mathrm{Call}_1, \mathrm{Call}_2, \mathrm{Call}_3\}$. These vectors are independent of each other and there is a respective support vector machine, random forest, and logistic regression model for each feature type as they cannot be classified with a shared model. When sufficient vectors of a given feature type are collected, the vectors are passed to our models for analysis, regardless of the status of other feature type vectors. This results in independent collection time and data sampling for each feature type. If all feature types return a benign classification for their respective vectors then a device is considered to be benign. If any of the vectors are classified as malicious, then the device is exhibiting suspicious behavior. It is worth noting that a malware may attempt to spoof one of its feature types such as its task structure but may not attempt to spoof all three feature types at the same time resulting in a more resilient IoT device.

2.0.7 Dual-Stage Classification

Dual-level classification architectures have been shown to be successful in classifying malware [XRSM17] [Yua17]. Besides classifying many aspects of a process, our framework uses a dual-stage approach. Our first stage is an anomaly detector which has been trained solely on benign data and is highly sensitive. Our second stage occurs if the sliding window is found to be anomalous after being analyzed by our anomaly detector (first stage). It is passed to our classifier which has been trained on both benign and malicious data. This classifier attempts a more in-depth analysis to further assess the window as benign or malicious and makes a decision. Our first stage can mark a task as suspicious, limiting its execution, while the second stage will perform a more in-depth analysis to fully determine the intent of the task. Figure 2.12 is a workflow diagram of our ensemble features using a dual-stage analysis approach. Our framework contains a one-class support vector machine, a random forest, and a logistic regression model for each of our feature types (task structure, system calls, and memory access patterns) which analyze independently of each other. A vector is first passed to the one-class support vector machine and if the vector is determined not to be benign, the vector is passed to both of our random forest and logistic regression models for comparison. The one-class support vector machine is trained solely on benign datasets and the random forest and logistic regression models are trained on the same dataset containing both benign and malicious samples.

Once a stage-one alert is identified the agent will be notified to extract more data and to monitor more features in an effort to pass the additional information to stage-two before final determination is taken. A Quasi-malware state is a state in which a device has limited performance and network impact while it in in an intermediate evaluation state. The agent on the deployed device can be informed after a stage-one alert in order to restrict or modify various aspects of the device (Quasi-malware state) prior to the stage-two determination in order to prevent further malicious behavior or the propagation of malware. If stage-two finds no malware the agent is informed to return to normal operations. Throughout the process a visual representation of the networked devices is maintained denoting the health of these devices. A key feature of our dual stage approach is its ability to be diverse enough to accommodate the varying degrees of configuration of deployed devices. This diverse nature enables our first stage to function with a different sliding window size and threshold than our second stage. The first stage can be configured to analyze a sliding window and threshold that has been shown to be overly sensitive while our second stage can be configured for accuracy.

2.0.8 Sliding Windows and Thresholds

Sliding window validation is the process of backtesting time series models by dividing the whole cross-sectional dataset into different training windows by specifying the width of these windows. The model is trained using the training window and applied the testing window in order to compute the performance of the first run. The training window slides to a new set of training records and the process is repeated

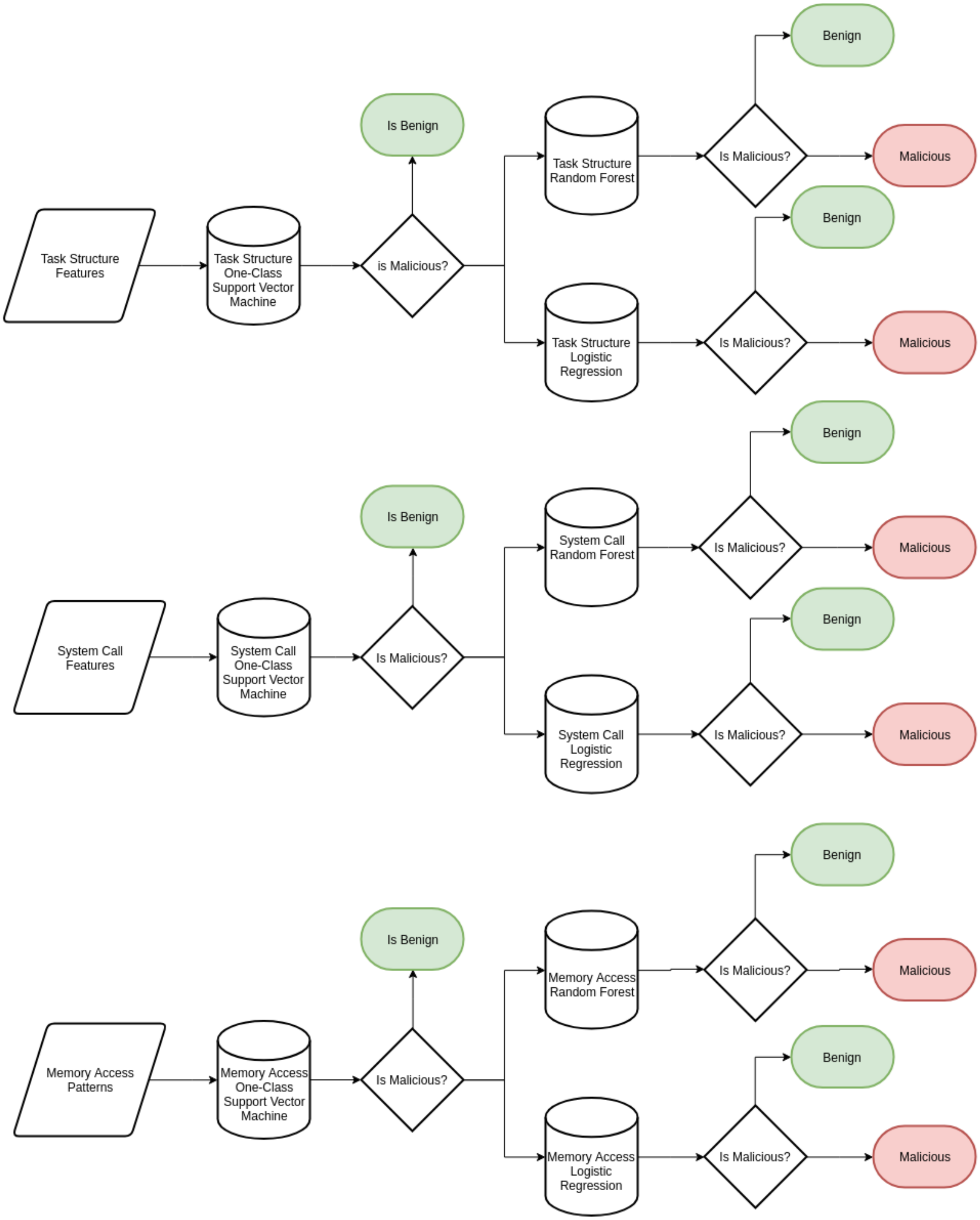

Benign
Is Benign
Task Structure Random Forest
Is Malicious?
Malicious
Task Structure Features
Task Structure One-Class Support Vector Machine
is Malicious?
Benign
Task Structure Logistic Regression
Is Malicious?
Malicious
Benign
Is Benign
System Call Random Forest
Is Malicious?
Malicious
System Call Features
System Call One-Class Support Vector Machine
Is Malicious?
Benign
System Call Logistic Regression
Is Malicious?
Malicious
Benign
Is Benign
Memory Access Random Forest
Is Malicious?
Malicious
Memory Access Patterns
Memory Access One-Class Support Vector Machine
Is Malicious?
Benign
Memory Access Logistic Regression
Is Malicious?
Malicious

Figure 2.12: Flow chart of ensemble approach including dual-stage classification

until all training windows are used. Using this technique for training, the average performance metric can be calculated across the entire dataset. The performance metric derived through the use of sliding window validation is generally more robust than split validation testing. Because of the improved robustness and accuracy of this mathematical technique, we implement sliding windows in the testing of our extracted features.

To improve the accuracy and robustness of our analysis, our framework implements sliding windows in its analysis of data. The values of our chosen features are harvested, combined into a vector for the current sliding window, and then transmitted to our framework. The sliding window is comprised of a set of vectors $\{V_1, V_2, \ldots, V_n\}$ where n determines the size of the window. Now, each vector V_x is composed of a set of features $\{F_1, F_2, \ldots, F_m\}$ where m denotes the last feature extracted from the defined feature set. The threshold within the sliding window is what percentage of the sliding window must be determined in order to make a determination for the entire window. For example a sliding window of size ten with a threshold of eighty percent requires eight of the ten vectors to be of a certain type before making the determination that the sliding window is of that type.

If we intent to use a sliding window of size ten for our task structure features we would include ten vectors, in other words ten rows of $\{F_1, F_2, \ldots, F_n\}$. For system calls with 3-grams it would be ten rows of $\{\mathrm{Call}_1, \mathrm{Call}_2, \mathrm{Call}_3\}$ and for memory access it would be ten rows of memory hits $\{\mathrm{Hit}_1, \mathrm{Hit}_2, \ldots, \mathrm{Hit}_n\}$. The sliding window sizes can also be independent of each other, in other words, task structure analysis can have a sliding window size of ten while system calls use a sliding window size of five. Figure 2.13 depicts a sliding window vector over time.

The accuracy of a sliding window correlates to the selected window size and the selected threshold. A window size that is too large with a low threshold may

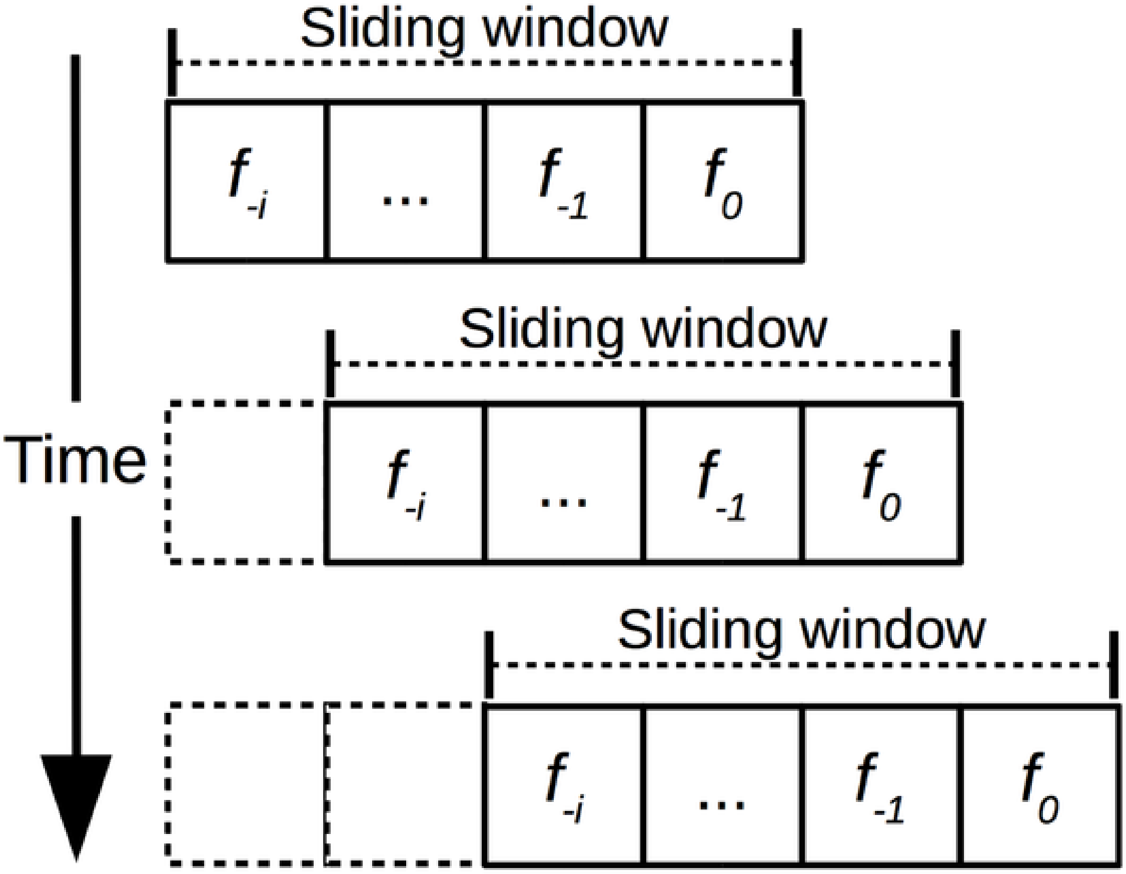


Figure 2.13: Attribute vector summarizing a sliding time window of frames

be too quick to make a determination for such a large window. This can result in inaccuracies especially with delayed malware which displays its execution later in the sliding window. Similarly a large sliding window with a high threshold must classify many vectors in order to make a determination. Classifying many vectors requires more processing time and such a high threshold may result in the determination being made late into the window. Comparably a sliding window that is too small may lack the required size to make an accurate determination of the devices running state. We have found that varying window sizes have different accuracy with various thresholds as shown in Figure 2.14 which demonstrates the accuracy for various sliding window sizes and thresholds when tested for our task structure.

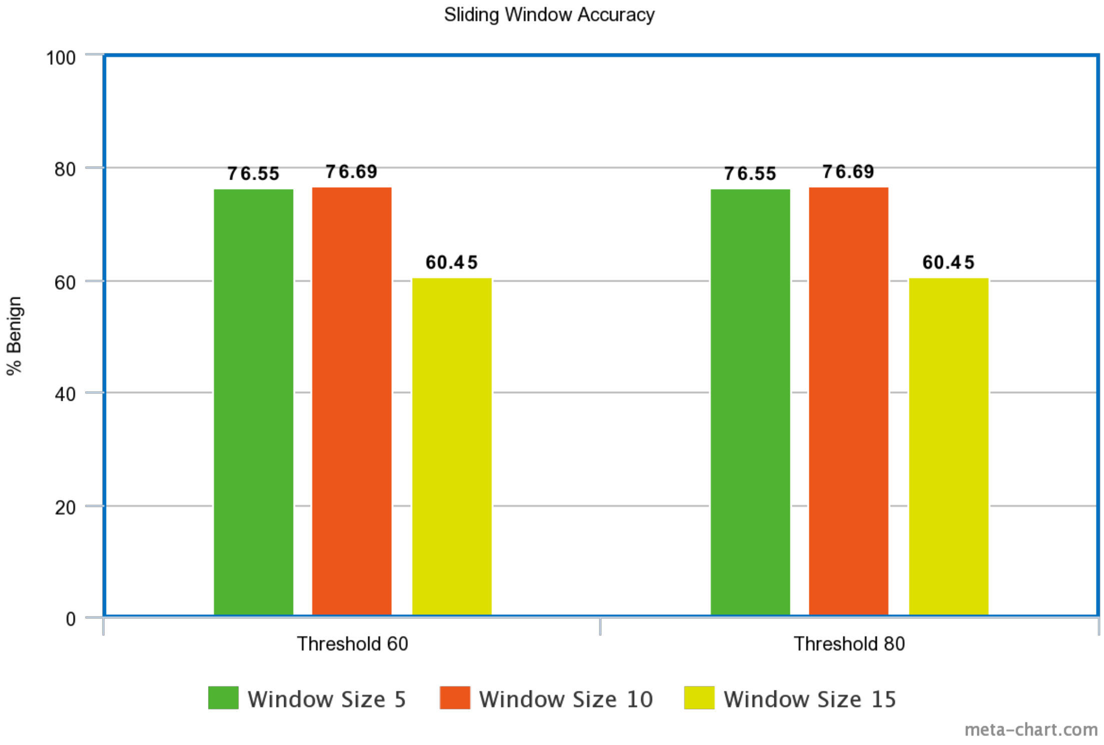


Figure 2.14: Accuracy of our various tested sliding window sizes and thresholds

2.0.9 Framework Validation

To validate our proposed framework we implemented our feature extraction within an x86 Linux environment and referred to [UGPL18b][UGPL18a] For the implementation of our framework the deployed devices not only have an embedded agent installed but the machine learning models would be trained at the manufacturer level. Before the device is deployed the devices would be run and analyzed in a secure environment and a map of their standard operating procedures would be developed. Our framework would be trained on this map for that device model so when the device is deployed it is ready to function with a model specific to it. In [UGPL18a] the authors showed success with various traditional machine learning algorithms such as support vector machines, random forest, and logistic regression. Because of this in our x86 Linux environment we used a one-class support vector machine as the model for our anomaly detection within our first stage. Within our second stage we used a random forest and logistic regression classifier to do our in-depth analysis.

A key feature of our dual stage approach is its ability to be diverse enough to accommodate the varying degrees of configuration of deployed devices and various classification algorithms. This allows a wide range of classification algorithms to be configured as many have been shown to have varying degrees of success for malware detection dependent on the devices environment [SBSF11] [Yua17] [JS17].

In order to combat malware, our methods of detection must improve. One possible approach is to use a dual stage classifier. The RansHunt framework shown in [Yua17] works on static and dynamic features and includes a multi-stage approach with a Support Vector Machine based algorithm. The framework is trained with 10-fold cross validation and uses Cuckoo Sandbox in the back-end for dynamic malware analysis. In the preliminary results, the Deep learning model with 2-tuple inputs achieved 94.83% accuracy. To improve detection time, system call traces were compressed and in all cases the RansHunt framework performed better than most anti-malware methods.

Similarly, [XRSM17] includes a two-level classification architecture which analyzes kernel rootkits versus user-level malware and uses n-gram attribute similarity (AS). Feature vectors of byte code are extracted and arranged into n-grams and tested against various algorithms for their accuracy and true positive rate (TPR). The researchers tested three data sets, the first of which was collected in 2014 and contained 88 malwares and 84 benigns. The second dataset is from 2015 and contained 200 malwares and 168 benigns while the third dataset consisted of 1000 malwares and benigns from the Open Malware Benchmark. The first data set was

Table 2.7: [XRSM17] Experiment Results of Dataset 2

	True Positive (%)	Accuracy (%)
AS	95.4	92.61
C4.5	90.7	91.30
Naive Bayes	89.95	91.00
BayesNet	91.65	91.66
SVM	82.15	88.78

Table 2.8: [XRSM17] Experimental Results of Dataset 3

	True Positive (%)	Accuracy (%)
AS	86.69	81.07
C4.5	58.64	59.7
Naive Bayes	37.63	68.31
BayesNet	52.15	74.67
SVM	42.9	70.93

used for training and the second and third datasets were used as test data, the results of which are detailed in Table 2.7 and Table 2.8.

CHAPTER 3

RESULTS AND DISCUSSION

Due to our ensemble approach, our framework relies on various machine learning models, specifically one-class support vector machine, random forest, and logistic regression. The one-class support vector machine is our anomaly detector and as such was trained solely on benign data. Our random forest and logistic regression models are our classifiers trained using 10-fold cross-validation and were trained on the same set of benign and malicious processes.

For task structure features, the random forest model had a model result mean of 99.98% and an accuracy of 100%. The task structure logistic regression model had a model result mean of 65.98% and an accuracy of 65.35%. Table 3.1 displays the classification report of the task structure random forest model while Table 3.2 displays the classification report of the task structure logistic regression model.

Table 3.1: Details of the Task Structure Random Forest model.

	Precision	Recall	F1-score	Support
Benign	1.00	1.00	1.00	481
Malware	1.00	1.00	1.00	890
Average/Total	1.00	1.00	1.00	1371

Table 3.2: Details of the Task Structure Logistic Regression model.

	Precision	Recall	F1-score	Support
Benign	0.50	1.00	0.66	471
Malware	1.00	0.47	0.64	900
Average/Total	0.83	0.65	0.65	1371

For system calls, the random forest had a result mean of 99.40% and an accuracy of 99.42% while the logistic regression had a result mean of 98.82% and an accuracy of 98.92%. The classification report for system calls of both the random forest and the logistic regression models can be found on Table 3.3 and Table 3.4, respectively.

Table 3.3: Details of the System Call Random Forest model.

	Precision	Recall	F1-score	Support
Benign	0.99	1.00	1.00	6143
Malware	1.00	0.46	0.63	67
Average/Total	0.99	0.99	0.99	6210

Table 3.4: Details of the System Call Logistic Regression model.

	Precision	Recall	F1-score	Support
Benign	0.99	1.00	0.99	6143
Malware	1.00	0.48	0.64	67
Average/Total	0.98	0.99	0.98	6210

For memory access pattern analysis, the random forest model displayed a result mean of 87.64% and an accuracy of 89.14%. The logistic regression model for the memory access pattern approach displayed a result mean off 87.67% and an accuracy of 89.26%. Table 3.5 demonstrates the classification report of the memory access pattern based random forest model while Table 3.6 demonstrated the logistic regression model.

Table 3.5: Details of the Memory Access Patterns Random Forest model.

	Precision	Recall	F1-score	Support
Benign	0.49	0.02	0.04	93
Malware	0.89	1.00	0.94	773
Average/Total	0.84	0.89	0.85	866

When classifying a process as a whole, on average across all three techniques, our one-class support vector machine had an accuracy of 86% served as our anomaly detector. Due to our dual-stage approach, the remaining 14% of unknown or malicious occurrences were passed to our random forest and logistic regression models for further classification at which point either model came to its own respective classification for the tested data. In total, our random forest and logistic regression models showed overall more accuracy than our one-class support vector machine

Table 3.6: Details of the Memory Access Patterns Logistic Regression model.

	Precision	Recall	F1-score	Support
Benign	0.40	0.02	0.03	93
Malware	0.89	1.00	0.94	773
Average/Total	0.80	0.89	0.84	866

model however required more time to classify the data. By using our anomaly detector as a first stage, we reserved in-depth classification and overhead to only the processes that truly required it.

Table 3.7 lists some benign and malicious processes which were used for the testing of our models. We attempted to use processes which produced a minimum of 1000 task structure vectors, 100 system call vectors, 100 memory access vectors. On average we have 500 task structure vectors, 500 system call vectors, and 200 memory access vectors per process.

Table 3.7: Example benign and malwares that were used to the accuracy of our framework

Tested Benign	Tested Malwares
Ack-grep, grep, bmon, cat, cbm, ethstatus,htop, ls, top, htop, wget, fzf, fortune, bc, task	Backdoor_c, dofloo, linux.worm.lupper, sotdas, kaiten

Due to our ensemble approach, our framework relies on various machine learning models, specifically one-class support vector machine, random forest, and logistic regression per feature type. Using the same malware and benign processes, Table 3.8 compares the accuracy of each of our individual model using the same dataset in order to demonstrate how our selected machine learning algorithms perform. These results are the averages of each feature type (task structure, system calls, and memory access) for each respective model.

Table 3.8: Comparison of the accuracy of our models for each of our extracted feature types

	Task Structure Accuracy	System Call Accuracy	Memory Access Accuracy
One-Class SVM	84%	93%	83%
Random Forest	98%	99%	89%
Logistic Regression	65%	98%	89%

Table 3.9 depicts the total average accuracy and speed of each of our machine learning model algorithms over all three feature types. We took our benigns and malwares and observed the average accuracy of each feature type for each given model. We further recorded the average speed it took to make a determination. The short time to determination we observed with the one-class support vector machine supports our use of that machine learning model as a first stage classifier as the first stage is required to be a rapid and robust anomaly detector. The increased accuracy of the random forest supports our use of it as a in-depth classifier.

Table 3.9: Average accuracy and speed of our models over all three techniques

	Average Accuracy	Average Milliseconds
One-Class SVM	85.67%	.0451
Random Forest	95.33%	.0858
Logistic Regression	84.51%	.0902

To improve upon our frameworks accuracy and robustness, rather than classifying a process as a whole, we implemented the use of sliding windows. These sliding windows consist of chunks of our extracted features of a process and are passed in similar fashion to our one-class support vector machine and anomalies are then passed to our two classification models. By classifying a process in chunks rather than as a whole, we were able to make earlier determinations as to whether a process was benign or malicious. Furthermore, delayed execution of malicious activity eventually fell into a sliding window as we iterated through the extracted features and registered as anomalous, increasing our chances of detecting malicious activity. The use of sliding windows also allows us to continuously monitor an executing application and classify smaller chunks of data rather than a process as a whole which results in less computation.

Each sliding window can also have a threshold with which judgement is made upon the window as a whole. For instance, a sliding window of ten records can be said to have a threshold of 80% if eight evaluated records are determined to be benign and thus the whole window determined to be benign. We tested varying window sizes while testing each with a threshold of percent benign to percent malware within each window. The sliding window sizes tested were five, ten, twenty, fifty, and one hundred. The thresholds tested were ninety, eighty, seventy, and sixty. This means that if the percent of benign vectors within the sliding window was determined to exceed the threshold then that whole sliding window is now classified as benign due to the prevailing percentage. An excerpt of this is shown in Figure 3.1 which depicts the varying accuracy depending on threshold for sliding window sizes five, ten, twenty, fifty, and one hundred. In other words, different sliding windows are more accurate with certain thresholds. The results depicted in Figure 3.1 are the averages from all three of our feature types (task structure, system calls, and memory access) using the same benign and malicious processes in Table 3.7 in order to test the overall accuracy of various sliding window sizes and thresholds. We tested our benigns and malwares using a sliding window of five with the various thresholds and averaged the results of our models. We repeated our tests over the same dataset but with different thresholds and window sizes in order for the dataset to remain our constant. Figure 3.1 demonstrates that the accuracy of a sliding window is affected by the threshold selected for that sliding window and that each sliding window must be carefully analyzed to determine its best accuracy setting.

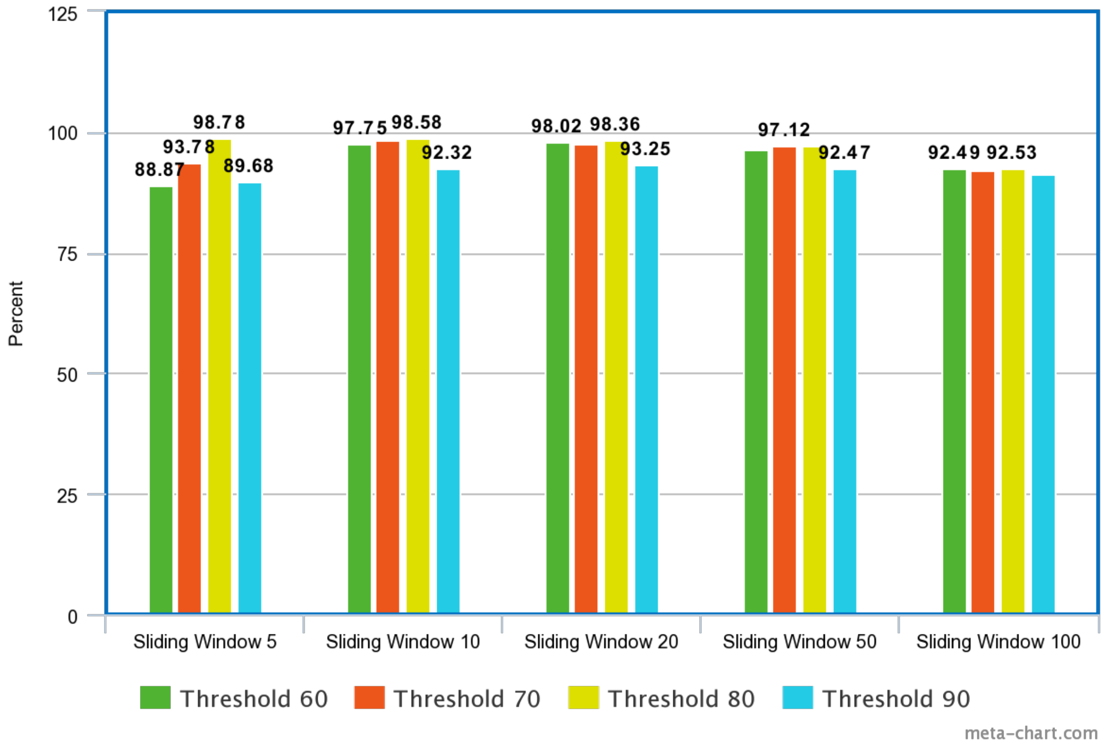


Figure 3.1: Threshold Comparison for Varying Sliding Window Sizes

Each of our individual models proved to be accurate in classifying a malicious process. Although our framework requires more training and testing, using probabilistic analysis, all three techniques are accounted for and a final determination can be achieved. Coupled with the use of sliding windows and a dual-stage approach for further accuracy, our framework shows great promise in determining when a device is compromised. Table 3.10 depicts the average accuracy of all three of our approaches for a given set of benign and malicious processes. Of note, the malware sotdas was determined to be unknown by all three of our models and for the security and integrity of our device, if a process if unknown or anomalous it is assumed to be malicious.

Table 3.10: Average accuracy of our models over our three feature types for two benign and three malicious processes

	Class	Ensemble OC-SVM	Ensemble Random Forest	Ensemble Logistic Regression	Average Detection
bmon	Benign	97.58% benign	96.72% benign	96.39% benign	Benign
ethstatus	Benign	98.42% benign	97.68% benign	96.33% benign	Benign
kaiten	Malware	89.85% unknown	93.94% malware	92.12% malware	Malware
linux.worm.lupper	Malware	87.37% unknown	91.94% malware	88.01% malware	Malware
sotdas	Malware	98.96% unknown	86.41% malware	85.73% malware	Malware

IoT devices tend to run a single CLI application on a dialect of Linux and as such, our focus was on CLI applications. However, we further tested our models against various popular graphical user interface applications within a our x86 Linux system such as Chrome a web browser, VLC an audio player, Gimp a photo editor, and Skype a communications platform to ensure our models were accurate when introduced to new applications. Our models exceeded our expectations and accurately identified each of these as benign. However, a model trained purely on command line interface (CLI) applications will have a more difficult time classifying a benign graphical user interface (GUI) application as benign due to the difference in behavior between a CLI and GUI application. For instance, GUI applications are event driven whereas a CLI application usually performs their function once executed. This inherent difference in behavior results in models requiring more training depending on their intended target application type, CLI or GUI.

Past research has shown that the individual use of task structure, system calls, and memory access patterns are viable and effective approaches to the detection of malicious activity [SBSF11] [CMK15b] [XRSM17]. Each individual approach has its benefits and merits, however we explore the possible synergy of these approaches to augment their benefits and increase the work required by a malicious actor to circumvent each technique. Furthermore, we present this ensemble approach as a dual-stage classification framework which uses a highly sensitive first stage and a more in-depth analytical secondary stage.

A promising factor of our approach is that it is dynamic enough to allow for different window sizes and thresholds per deployed device for further usability in distributed networks that run varying device models. In other words, device types will have their own dual-machine learning system, customized to their respective window size and threshold value.

To further improve our framework, we propose exploring other available task structure features and system calls which can be used to train our model. Android based devices should also be tested [XLQZ12] as well as windows based operating systems [UGPL18a] and a complete cloud based solution for intrusion and compromise detection [HZB11].

CHAPTER 4

CONCLUSION

Malware is proving to be more dangerous than ever before and with the ubiquity of deployed IoT devices it is increasingly difficult to determine which devices have been maliciously compromised. We propose a framework which uses a dual stage behavioral analysis approach with increased accuracy of detection by combining the observance of task structure features, system calls, and memory access patterns into an ensemble approach. Coupled with a dual-stage anomaly detector and classifier, our framework offers high accuracy and malware detection to determine the integrity of an IoT device. These approaches show success on a variety of devices such as Windows, Linux, and Android. Various algorithms such as Support Vector Machines, Random Forest, and Logistic Regression have been tested with varying yet promising accuracy when compared to Naive Bayes and Nearest Centroid algorithms.

Furthermore, these machine learning based malware detectors have been shown to require more work by a malicious actor in order to circumvent their detection. Our model showed great accuracy with limited training data. This approach allows a device to contain an embedded agent and extract key process data as the device executes its tasks. Furthermore, the data could be transmitted as a collected vector to a cloud framework for processing.

Within a multi-stage framework, the first, sensitive, anomaly detection stage flags malicious executions and passes them to a secondary stage for a more in-depth analysis and classification. Once the malicious activity is detected a device can be halted or the task suspended from functioning within the network in order to prevent further execution. Furthermore, once refined, this approach may work with little overhead and interruption on the end device. This near real-time detection would detect various kinds of malware and could stop them before they complete their execution even when a malware attempts to delay its execution to obfuscate itself.

Our approach shows promise as it can be combined with other approaches for more accuracy and a higher detection rate while also maintaining low battery and CPU costs. It is our hope that as the rapid evolution of technology in our sociality increases so does our defensive and countermeasure capabilities. Malware is attributed as the cause of massive data and integrity loss. Due to the dynamic and aggressive nature of Malware we require dynamic, vigorous, and progressive approaches to defend against such attacks.

BIBLIOGRAPHY

[AHHT13] A. S. A. Aziz, A. E. Hassanien, S. E. Hanaf, and M. F. Tolba. Multilayer hybrid machine learning techniques for anomalies detection and classification approach. In *13th International Conference on Hybrid Intelligent Systems (HIS 2013)*, pages 215–220, Dec 2013.

[BSM13] K. Blokhin, J. Saxe, and D. Mentis. Malware similarity identification using call graph based system call subsequence features. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 6–10, July 2013.

[CCL+09] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 555–565, New York, NY, USA, 2009. ACM.

[CLM01] João B. D. Cabrera, Lundy Lewis, and Raman K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. *SIGMOD Rec.*, 30(4):25–34, December 2001.

[CMK15a] R. Canzanese, S. Mancoridis, and M. Kam. Run-time classification of malicious processes using system call analysis. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 21–28, Oct 2015.

[CMK15b] R. Canzanese, S. Mancoridis, and M. Kam. System call-based detection of malicious processes. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 119–124, Aug 2015.

[DA13] E. Dorj and E. Altangerel. Anomaly detection approach using hidden markov model. In *Ifost*, volume 2, pages 141–144, June 2013.

[DGSTG09] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 566–577, New York, NY, USA, 2009. ACM.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the*

1996 IEEE Symposium on Security and Privacy, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.

[FT17] Z. Fuyong and Z. Tiezhu. Malware detection and classification based on n-grams attribute similarity. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 793–796, July 2017.

[HZB11] Amir Houmansadr, Saman A. Zonouz, and Robin Berthier. A cloud-based intrusion detection and response system for mobile phones. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 31–32, Washington, DC, USA, 2011. IEEE Computer Society.

[JMJ18] M. Jaiswal, Y. Malik, and F. Jaafar. Android gaming malware detection using system call analysis. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–5, March 2018.

[JS17] A. Jain and A. K. Singh. Integrated malware analysis using machine learning. In *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, pages 1–8, Aug 2017.

[LHC11] S. Liu, H. Huang, and Y. Chen. A system call analysis method with mapreduce for malware detection. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 631–637, Dec 2011.

[RRL$^+$14] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu. Data-centric os kernel malware characterization. *IEEE Transactions on Information Forensics and Security*, 9(1):72–87, Jan 2014.

[RTWH11] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, December 2011.

[SBSF11] F. Shahzad, S. Bhatti, M. Shahzad, and M. Farooq. In-execution malware detection using task structures of linux processes. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–6, June 2011.

[SN08] L. Sun and T. Nakajima. A lightweight kernel objects monitoring infrastructure for embedded systems. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 55–60, Aug 2008.

[UGPL18a] H. Upadhyay, H. A. Gohel, A. Pons, and L. Lagos. Windows virtualization architecture for cyber threats detection. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 119–122, April 2018.

[UGPL18b] Himanshu Upadhyay, Hardik Gohel, Alexander Pons, and Leo Lagos. Virtual memory introspection framework for cyber threat detection in virtual environment. *Advances in Science, Technology and Engineering Systems Journal*, 3:25–29, 01 2018.

[XLQZ12] K. Xin, G. Li, Z. Qin, and Q. Zhang. Malware detection in smartphone using hidden markov model. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 857–860, Nov 2012.

[XRSM17] Z. Xu, S. Ray, P. Subramanyan, and S. Malik. Malware detection using machine learning based analysis of virtual memory access patterns. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 169–174, March 2017.

[XZSZ10] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. Pbmds: A behavior-based malware detection system for cellphone devices. pages 37–48, 03 2010.

[Yua17] X. Yuan. Phd forum: Deep learning-based real-time malware detection with multi-stage analysis. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–2, May 2017.

[ZLK$^+$07] K. Zhao, Q. Li, J. Kang, D. Jiang, and L. Hu. Design and implementation of secure auditing system in linux kernel. In *2007 International Workshop on Anti-Counterfeiting, Security and Identification (ASID)*, pages 232–236, April 2007.

[ZZSL15] K. Zhao, D. Zhang, X. Su, and W. Li. Fest: A feature extraction and selection tool for android malware detection. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 714–720, July 2015.