2017-04-21

# Data-Driven Malware Detection Based on Dynamic Behavioral Features

Rui Han
*University of Miami*, iurnah@gmail.com

UNIVERSITY OF MIAMI

# DATA-DRIVEN MALWARE DETECTION BASED ON DYNAMIC BEHAVIORAL FEATURES

By

Rui Han

A DISSERTATION

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Coral Gables, Florida

May 2017

UNIVERSITY OF MIAMI

A dissertation submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

# DATA-DRIVEN MALWARE DETECTION BASED ON DYNAMIC BEHAVIORAL FEATURES

Rui Han

Approved:

Jie Xu, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Mohamed Abdel-Mottaleb, Ph.D.
Professor of Electrical and Computer Engineering

Mei-Ling Shyu, Ph.D.
Professor of Electrical and Computer Engineering

Michael Scordilis, Ph.D.
Associate Professor in Practice of Electrical and Computer Engineering

Saman Zonouz, Ph.D.
Assistant Professor of Electrical and Computer Engineering
Rutgers University

Guillermo Prado, Ph.D.
Dean of the Graduate School

HAN, RUI (Ph.D., Electrical and Computer Engineering)

Data-Driven Malware Detection Based on Dynamic Behavioral Features (May 2017)

Abstract of a dissertation at the University of Miami.

Dissertation supervised by Professor Jie Xu.
No. of pages in text. (130)

Malware programs, such as viruses, worms, Trojans, etc., are a worldwide epidemic in the digital world. Studies and statistics show that malware volume has increased tremendously year after year in the past decade. Due to the rapid malware growth in recent years, the malware detection approaches have been experiencing a paradigm shift from the laborious manual analysis, signature-based approach to a data-driven, machine learning-based approach.

This thesis presents a semi-automated malware detection solution using machine learning. It notifies the user if the application she downloaded behaves differently than what she expected at download time. The hypothesis is that in spite of millions of currently downloadable executables on the Internet, almost all of them provide functionalities from a limited set. Additionally, because of each functionality, e.g., text editor, requires particular system resources, it exhibits a unique system-level activity pattern. During an on-line training process, the system creates a profile dictionary of various functionalities. This profile dictionary is then used to warn the user if she downloads an executable whose observed activity does not match its advertised functionality.

The proposed solution is deployed as a cloud service. It includes a multi-model classification module that takes into account the time-variant property of functionality and behavior features from the system level. Since static features are easier to be extracted, but it is less effective compared to dynamic behavioral features; Dynamic behavioral features are much more pricey to collect, but it is very effective. However, the effectiveness of dynamic behavioral features depends on the length of analysis; thus accurate detection requires more time and computing resources. Existing works focused on improving the model accuracy by discovering distinctive features in static analysis or dynamic analysis. Despite these recent advances, to implement an efficient and user interactive malware detection system remains challenging. The uniform length of dynamic analysis adopted by previous research failed to capture the ongoing evolvement of malware behaviors. Extending the duration of dynamic analysis, although advantageous in improving the accuracy, is nevertheless both resource intensive and time-consuming. There exist a need to balance the accuracy and resource consumption in a practical system. We modeled the system using contextual multi-armed bandit framework and presented two on-line learning algorithms that, for each sample to be analyzed ensures the high probability of selecting the best classifier. To that end, we define Quality of Experience (QoE) as a user metric in the framework to balance the accuracy and efficiency trade-off and use static file feature as the context to facilitate the classifier selection. Our experiment results using 2000 real malware samples show that context specification of classifiers can be discovered over time to create a strong detector given K weak detectors.

*to my families*

# Acknowledgements

First and foremost, I would like to thank my research advisor Dr. Jie Xu and Dr. Saman Zonouz for their support and guidance in the past few years. I feel very fortunate to have had the chance to work with Dr. Jie Xu and Dr. Saman Zonouz. They are always supportive and provides me with extremely helpful advice both for my Ph.D. research and beyond. I thank Dr. Mohamed Abdel-Mottaleb, Dr. Mei-Ling Shyu, and Dr. Micheal Scordilis for their advice to my research and the inspiring technical discussion we had.

I also thank my internship manager Dr. Kevin Qin and Hayley Zhang at Juniper Networks for sharing their perspectives in career choices. Additionally, many thanks go to Hopper Wang, Xiaosong Yang, Yong Hao.

I thank my dear parents Ping and Jianye, and my lovely sister, Yu for their constant support and encouragement. It would have been impossible for me to go this far without their support.

Last but not the least, I am grateful to my girl friend, Da Zhang, whose support, inspiration and love was absolutely essential for success of the work I have done.

Rui Han

*University of Miami*

*May 2017*

# Table of Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

The number of cyber attacks has been skyrocketing in the past decade. The significance can be verified by the high frequency of headlines that cover cyber attack incidences around the world. The reason is twofold: more and more people and businesses hold social occasions and make business transactions digitally; and less and less effort is needed to successfully carry out a cyber attack because of the emerging attack generation toolkits. Malicious software, namely malware, is one of the major root causes of everyday cyber attack incidences. According to a security report [2], in 2014, 15% of 40 million files monitored in several corporate networks are malware. The growing proliferation of malware is profoundly detrimental to the general public, businesses, and national security. It poses threats to the integrity and security of personal data and computer systems. Malware infiltrations could result in various consequences, such as personal or corporate data breaches and critical infrastructure failures. To defeat cyber attacks brought by malware, defenders should be able to identify and block the malware from the host computer or network traffic. In Section 1.1, we will first review some classic methods in malware detection and summarize the main challenges. In Section 1.2, we give a summary of the major contributions of this thesis. In Section 1.3, we outline the structure of this thesis.

## 1.1 Motivation

Traditionally, malware detectors use virus signature scanners and heuristic methods to defend against malicious software. Most current commercial anti-virus tools rely on a database of syntactical patterns or regular expressions that characterize known malware variants. Anti-virus companies very often update their databases whenever an unknown malware variant is encountered in the wild. File features, with all their different formats, data structures, and metadata, are the origin of most anti-virus signatures. To generate signatures from these file features demands significant human works. Although a practical solution to prevent the known malware samples from spreading, signature-based malware detection fails to detect new malware samples and new malware variants. Moreover, malicious files can be either packed or encrypted to avoid being analyzed or can be encapsulated with obfuscated attack code that deceives. For some cases, the malicious files conceal their maliciousness until triggered by certain system events or by downloaded payloads. With these challenges, the signature generating rate fails in catching up to the malware production rate. Many researchers have proposed to share signatures between different organizations to combat the overflow of newly created malware. The cooperation rarely happens because the anti-virus industry treats malware signatures as intellectual property and refuses to share them. Even if companies are willing to share, the proprietary formats or specifications will render such interaction impossible.

In aware of the drawbacks in signature based detection, many recent studies applied a data analysis method such as machine learning in malicious file detection. The hypothesis behind the method is that if one can extract distinctive features from a large amount of labeled malicious and benign samples, with the help of cross-validation strategies, one can train an accurate classification model for the known malware samples. The obtained model could be applied to predict the maliciousness of new malware samples. The challenge of the method primarily lies in the feature extraction step. A multitude of works have studied static file features in applying machine learning to achieve automatic malware detection [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. In this literature, the static file features are obtained solely from the file contents without executing the file itself. However, it has been shown that static feature based malware detectors are susceptible to evasion attacks [15], [16], [17]. For example, Xu et al. applied generic programming algorithms in mutating malicious samples to generate evasive variants and achieved 100% successful evasion for the specific detector studied in the research [17].

Instead of analyzing the features based on the static file content, dynamic analysis takes the recorded runtime activities as features. The procedure is to run the file, i.e. a Windows executable, in an instrumented environment and record all the runtime behaviors that can be analyzed by cluster or classification algorithms. Malware detection based on dynamic analysis received a lot of attention in industry because of the growing need to automate the malware analysis procedure. Take VirusTotal [18] as an example, it executes malware samples in a controlled environment and monitors their behaviors. Reports are generated from the execution traces to support a malware analyst to reach a conclusion about the level of severity imposed by a malware of the threat. To completely free the human analyst from reading the tedious reports, Rieck et al. built Malheur, a machine learning based malware clustering tool based on behavior reports [19]. Bayer et al. successfully clustered millions of malware samples into groups by applying various data analysis methods in generated reports [20], [21], [22]. As malware runtime behavior features are harder to be modified to create mimicry attacks compared to static file content features, machine learning based method with dynamic features is superior to a method with static features because it is harder to be evaded. However, the dynamic feature extraction requires more complex system configuration and higher resources consumption.

In building dynamic behavior feature based detection system, previous work set forth some promising systems and demonstrated the detection effectiveness using private sample collections with individually engineered behavior features. Nevertheless, these conventional design have some inherent drawbacks, without being resolved, will prevent them from being deployed in an effective and user interactive environment. First and foremost, existing works trained models using partial behavioral features collected from a heuristically determined period of dynamic monitoring, usually within less than 2 minutes from the start of execution. There is no performance guarantee that the selected time length is close to optimal and is also applicable to future analysis without performance degradation. For a certain batch of binary samples, one might find adding a extra minute in behavioral tracing could increase the detection accuracy. More importantly, predetermined execution time exposes the detection system to deceptive malware attacks, which would take preemptive delay of the malicious activity against the detection. Moreover, the determined execution length usually applies to all the samples that will result in waste of computing resource. For instance, if a malware sample reveal its malicious traits in the first few seconds, whereas the system default tracing length is 2 minutes. There is lacking an analytical method to determine the execution length for different samples dynamically.

Second, the authors of conventional approaches trained their classifier using cross validation method and evaluate it with limited testing sample from past, which could ruled out other better classifiers had they trained with longer behavior features and evaluated with more samples. As a consequence, the system will perform significantly worse in the long run because it gets stuck on the suboptimal classifier. From the perspective of reinforcement learning, existing works are exploiting the best classifier evaluated using the samples at hand without exploring other classifiers that may performance better in the long term. The trade-off between exploring new classifier and exploiting the best classifier identified by the moment need to be carefully balanced to achieve more accurate detection results.

Third, they focus only on achieving high accuracy without considering the cost of the system, especially the resource usage and time cost of dynamic execution of the samples. Ignoring the cost make those systems less scalable or even impractical to be implemented as an interactive malware detection solution. In an endpoint system, it is generally accepted that user experience is affected by both the accuracy and the waiting time. Increasing the time of dynamic feature extraction in conventional system can explore more traits of malware behaviors which could improve the detection accuracy, but longer waiting will bring fair quality of experience for user and pose heavy system load for the server. To provide better quality of experiences, the resource constrain need to be take into consideration together with the accuracy requirements.

The proposed system bridge the gap between the existing dynamic behavior feature based machine learning method and a user interactive malware detection system with high quality of experience. We take into account the fact that malware features collected from different execution length will incur different time and resource cost, and produce classifiers with different accuracy. Specifically, longer execution will explore the behavioral feature space extensively and consume extremely high computation resources. In contrast, short binary execution period will only exercise a limited dynamic feature space thus cost less in resource usage. We introduce a online learning approach to select a particular classifier for each submitted file based on the classifiers' history Quality of Experience (QoE) measurement and the context of the file sample. The system is modeled using contextual multi-armed bandit framework to balance the exploitation and exploration of the available classifiers. Since these classifiers are trained with behavior features from distinct tracing periods, the selected classifier for next classification task reflect how long the sample should be executed. The determined analysis length could be used to notifying the user of the needed waiting time. To facility the multi-armed bandit learning, we explore the similarity information among the samples' context features (structural file features). We proposed two efficient on-line learning algorithms that learns over time the best mapping from context features to the best matching classifier with the QoE metric. The QoE provide a knob to allow the system to adjust the trade-off between accuracy and resource usage under different use cases.

## 1.2 Contributions

We addressed several of the challenges in detecting socially engineered malwares and in improving the performance of machine learning based malware detection system. We proposed a novel system TroGuard, which is capable of analyzing the functionality of the software based on its system level activities. To that end, we defined functionality classes according to user's perception and rule-based mapping that cast system level dynamic activities (syscalls) to intermediate behavioral features. Supervised learning algorithms is applied afterward to bridge the gap between the functionality classes and the behavioral features. To better improve user's quality of experience. A cloud version of TroGuard is proposed and modeled using contextual multi-armed bandits framework to balance the trade-off between the detection accuracy and the corresponding detection time and computing cost. We have built the prototypes and successfully evaluated the proposed approaches. To sum up, we make the following contributions in this thesis:

- A new approach to detect socially engineered malware downloads, that at its core relies on comparing the functionality expected by the user (efficiently determined through in-depth and automated analysis of the download web site) to the functionality exhibited during the downloaded application's execution;
- An end-to-end system for detecting Trojan downloads is designed and implemented to identify mismatches between user-perceived functionalities and actual software functionalities. TroGuard consists of a browser plugin and a host monitoring tool that communicate with each other to bridge the gap between the user experience and system activities;
- A comprehensive modeling of the cloud version of TroGuard using contextual multi-armed bandit framework and algorithms run in the framework to optimize the quality of experience of users and reduce the dynamic analysis cost.

## 1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces the related work in machine learning based malware detection and collaborate malware sharing platforms. Chapter 3 presents a host-based machine learning-based malware detector to defeat socially engineered Trojan. Chapter 4 describes the cloud system architecture of the enhanced malware detection system and the multi-armed bandit model to optimize the overall detection performance, characterized by Quality of Experience (QoE). Chapter 5 presents the empirical study of the effectiveness of static and dynamic feature based malware detection. Finally, Chapter 6 presents concluding remarks and discusses some possible directions to extend this work in future.

# CHAPTER 2

# Related Work

## 2.1 Traditional Method Against Malware Threats

Safe Browsing, a commercial service of Google, protects devices by showing warnings to users when they attempt to download dangerous files [23]. It detects malicious software downloads by matching URLs against the system‘s constantly updated lists of malicious URLs. In practice, update cycle of Safe Browsing database may cause inherent delays in protecting users against frequently mutated URL and binaries. In addition, many deceptive behaviors currently pervasive to software bundling could disarm Safe Browsing's protection when paired with deceptive promotional tools or social engineering tactics. Abu Rajab et al. presented a binary reputation system CAMP [24] that hardens the original Safe Browsing design. CAMP use binary circuits to compute software reputation scores from several statistical attributes derived from client request features and server side dynamic analysis features. Since CAMP is a proprietary service, there is no open knowledge about the design of the server side dynamic analysis module. We believe it suffers from the same drawbacks as those study based on dynamic analysis which will be discussed in the following section.

## 2.2 Machine Learning Based Malware Detection

Attackers often choose file based exploits to gain access to the victim system because files can be reliably delivered to intended targets and easily modified to avoid signature detection. To defeat file based intrusions, defenders should be able to detect, quarantine, and remove malicious files from various digital media such as hard disk or network traffic. Signature based detection method an practical solution to prevent the known malware samples from spreading, fails to detect new malware samples or mutations. Counter measures such as machine learning based methods have been studied extensively and implemented in some real world intrusion detection systems. Kong et al. [12] classify malware variants into corresponding families by extracting function call graphs and using it as malware features. The work adopts an ensemble classifier and applies it in a dataset containing unpacked malware instances from 526,197 unique files. The experimental result of the work achieves the best F1 score of 98%. In other's work, N-gram byte-level file content have been used as detection features and achieved high accuracy in [3], [4], and [5]. Unfortunately, this n-gram model have been demonstrated vulnerable to evasion attacks by Wagner and Soto in [6]. Other literatures explore edit distance between instruction sequences [7], [8], [9] and graph representations of disassembled executable code [10], [11], [12] as detection features. Recently, there is a stream of work on maclious PDF file detection which focuses on applying machine learning methods and extracting file features from structure pdf data and meta data [13], [14], [15], [16], [25], [17].

The series of aforementioned literatures on malicious PDF detectors have thoroughly studied the method applying machine learning in achieving automatic malware detection from both attacker and defender's point of view. The feature vectors used in these works are mainly based on PDF's moderate complex structural features. Smutz et al. [13] extract features from document meta data and structure information, such as "number of characters in the title" and "the size of the images in the document". Although these features are not explicitly measuring inherently malicious attributes, it is proofed by experimental dataset that they are reflective of malicious activity and are useful in identifying malicious PDF files. Their final prototype, PDFrate[1] is freely open to public for malicious PDF detection research and evaluation. They evaluate PDFrate's robustness against direct evasion and mimicry attacks. To generate evasion samples, the author use normally distributed data to perturbate a subset of most important features extracted from malicious PDF files. The experiment result shows that the evasion rate is relatively low and PDFrate is resilient to the type of evasion attacks in which a malware sample mimics the important features of a benign file in order to be classified as benign.

Šrndić et al. [14] make use of the difference of structure paths in malicious and benign PDF documents to separate them apart, thus to achieve detection purpose. They parse PDF files according to the PDF Reference specifications, convert the logical document structures of various PDF objects into the structure path format, extract features by counting the structure paths, and finally train and evaluate a classification model based on the vectorized structure path counting. The experiment results evaluated on several heterogeneous dataset show that the PDF structure path feature could be very effective when using decision tree and SVM classifier. At last, they discuss the possible techniques to evade their method. According to their reasoning decision tree classifier achieves high accuracy in experiment study, but it could be easily evaded due to simplicity. To attack a decision tree classifier, attacker

[1] https://csmutz.com/pdfrate/

can build decision tree and generate evasive samples by trial and error. Similarly, SVM with linear kernel is susceptible to the similar evasion attacks. However, SVM with RBF kernel, which has the best performance as it demonstrated by experiment result, is almost impossible to evade. Additionally, they also presents an experiment of 10 weeks continuous testing on an accumulated dataset and concluded that the detection system outperforms most common commercial antiviruses.

The collection of detection and evasion studies on the infamous machine learning based malicious pdf file detector PDFrate have showed that, despite machine learning based method can achieve high effectiveness in malicious pdf file detection, such a system will not be able to use reliably in practice until an algorithm which is resilient to mimicry attack is proposed. For example, the evasion work by Xu et al. [17] apply generic programming algorithms in mutating malicious samples in order to generate evasive variants. Without knowledge about the classifier and the training data, their research experiments achieve 100% evasion rate.

As malware runtime behavior is much more difficult to modify in order to create mimicry attacks compared to static file contents features. Malware detection based on dynamic analysis receives a lot of attention [19], [20], [21] [22]. Instead of analyzing the features based on the static file contents, dynamic analysis look into the runtime activities. The idea is to run the file, i.e. an executable, in an instrumented environment and record all the runtime behaviors for being analyzed by clustering or classification algorithms.

Bayer et al. [20], [21], [22] studied dynamic malware behavior clustering system. Their method is to execute malwares in a sandbox environment that monitors the system call traces and taint input data to keep tracking how the malware manipulate and move the data. Consequently, a complete malware behavioral profile is generated in terms of operating system objects, operating system operations that are carried on those objects. Their work also includes network analysis which extracts high-level semantic operations from low-level socket system calls such as names of download files, name of IRC channels, or mail subjects. To achieve higher scalability, they proposed a fast clustering algorithm based on locality sensitive hash to reduce the complexity of distance calculation from $n^2$ to $n^2/2$. A reference clustering dataset has been used to evaluate efficiency of the algorithm, precision and recall reported together with previous methods show that their behavior profile and clustering method are superior to the existing works. Lastly, they also evaluate the scalability and efficiency of their solution on test set include 75,692 samples. It takes total of 2 hours and 18 minutes to complete the clustering task and keep memory usage under 4 GB.

Rieck et al. [19] and Trinius et al. [26] proposed a malware detection framework that automatically analyze dynamic malware behaviors using machine learning. The system is able to identify new class of malware by behavioral clustering and to classify new malwares into already identified classes. The project also propose a incremental approach for behavior-based analysis to process thousands of binaries on a daily basis, thus to reduce the analysis time significantly. The first novel point is the **Malware Instruction Set (MIST)**. Which is a format that record dynamic behaviors. For example, each MIST instruction encodes one monitored system call and its arguments. An the data encoded in the instruction are arranged in different level of blocks, reflecting behavior with different degree of specificity. Those arguments are encoded as a table index, which will map to the original contents, such as file and mutex names.

It describe the behavior of malware using “Q-grams”. Embedding function $\phi(x)$ is used to vectorized the MIST, then $\phi(x)$ will be normalized to $\hat{\phi}(x)$. The distance measurement is based on $\hat{\phi}(x)$, and Euclidean distance is measured. The clustering and classification are based on the distance measurements. To solve the scalability issue, "prototypes" are first extracted from the data, they are basically reports being typical for a group of homogeneous behavior. The paper illustrate the algorithms to do prototype extraction, clustering using prototypes, and classification using prototypes in order. Lastly, an incremental analysis algorithm is proposed to repeat the above algorithms on new data collected.

Sommer et al. [27] have studied the differences between network intrusion detection and other areas where machine learning finds successful. The paper discussed challenges of applying machine learning in intrusion detection. It conclude that the "strength of machine-learning tools is finding activity that is similar to something previously seen, without the need however to precisely describe that activity up front." While this paper is quite negative, it only limited in the static features. Pattern from network traffic could be see as static feature, while we would focus on features obtain from dynamic analysis. [28] is a enterprise solution that take two dimension features (byte content of image), and pass through deep neural network designed for image to detect the abnormal of malwares samples.

In our preliminary work, which will be presented in Chapter 3 in detail, we present TroGuard, a semi-automated web-based trojan detection solution, that notifies the user if the application she downloaded behaves differently than what she expected at download time. TroGuard builds on the hypothesis that in spite of millions of currently downloadable executables on the Internet, almost all of them provide *functionalities* from a limited set. Additionally, because each functionality, e.g., text editor, requires particular system resources, it exhibits a unique system-level activity pattern. During an offline process, TroGuard creates a profile dictionary of various functionalities. This profile dictionary is then used to warn the user if she downloads an executable whose observed activity does not match its advertised functionality (extracted through automated analysis of the download website).Our experimental results prove the above mentioned premise empirically and show that TroGuard can identify real-world socially engineered trojan download attacks effectively.

Although the accuracy is impressive in the experimental dataset, the above discussed solutions are likely to fail in real adversary scenarios when exposed to a broader scope of attackers and attack mechanisms. On one hand, To avoid detection, malware authors managed to generate mimicry malware by hiding obfuscated malicious code in a seemingly legitimated binary. They entice user to download such a malware-bearing file by various social engineering scams. Those mimicry malware are usually highly evasive. On the other hand, because the capabilities of an individual IDS system is limited by its dataset size no matter what classification algorithm is selected. Therefore, automatic and accurate malware detection requires not only advanced file feature analysis and dynamic behavior based detection, but also high collaboration on sharing malicious files and indicators.

In face of the burden to effectively detect new malwares samples, it is a natural reaction to aggregate multiple detectors to achieve a higher accuracy. Researchers proposed the idea of collaborative malware detection in [29,30] and designed malware analysis systems to facilitate collaborative detection efforts. The landscape of sharing thread intelligence has expand ever since. There exist more than a dozen of free, automated malware analysis services [31] online. Some of those have been highlighted in research paper such as VirusTotal and Anubis. Those services can examine the uploaded sample dynamically and generate a detailed activity report. Whether they are static or dynamic analysis based, above mentioned malware detection services are lack of data analysis capabilities for the generated report.

## 2.3 Executable Behavior Profiling

Lo et al. [32] design a system for finding program bugs or abnormal code runs caused by intrusions. In their terminology, a software behavior is a series of events including execution of a statement, a method call, or a basic block in a control flow graph. The technique is not scalable enough for real time use, in contrast to TroGuard.

In their propose frame work, a scalable algorithm is proposed to mine closed unique iterative patterns from program traces of known normal and failing executions. Following this pattern mining step, highly discriminative patterns is selected and a classifier is constructed based on the training traces with such pattern-based feature. Although the behavior features in [32] is accuracy enough in some software reliability data set, which only include two classes, the reliable software traces and unreliable software traces. Those features will fail our malware detection technique if we apply them as our feature list. Since the iterative pattern in software is not discriminative enough for a large amount of software classes. With respect to our higher level behavior features, adding those software iterative patterns into our feature list will overlap or breach the scalability of our method. Because it is possible that two different type of applications may have similar iterative patterns in their program execution traces.

Okazaki et al. [33] introduced early their work on process profiling from system call traces to deploy an anomaly-based intrusion detection system. The profile is simply a record of system call types and their observed frequencies. Although efficient, their approach is not resilient to circumvention by attackers, who can easily design their trojan to perform any required number of system calls to match a desired profile. Zhang [34] proposed a system to improve the resource allocation in grid computing based on whether the application belongs to CPU intensive, I/O and paging intensive, network intensive, or idle. Although they demonstrate their work improved the throughput in schedule tasks in clusters, it is unclear that the same characteristics are useful in a security setting.

## 2.4 Reinforcement Learning Based Approach

To the best of our knowledge, no existing research studied malware behavioral classification and detection using reinforcement learning. We reviewed papers that applied contextual multi-armed bandits framework in activity classification, and cloud computing based face recognition application. Xu et. al. have proposed a real time, context driven activity classification system that is capable of learning the best activity classifiers to accurately classify human activity data from wireless wearable devices [35]. Context information such as location of the activity and user's personal profile including genders, age, weight, etc. have been use to assist the learning. The proposed learning algorithm exploited the context similarity and adaptively partitioned the context space into smaller subspaces and learn the best oracle classifiers within each subspace. The simulation results of their demonstrate that proposed system and algorithm have significantly improved the performance of activity classification and reduce the energy consumption for power critical wearable devices. Onur et. al. studied contention and congestion problem in mobile device based graphic pattern recognition via cloud computing infrastructure [36]. They proposed two multi-armed bandits algorithms to learn the optimal condition for image transmission and recognition in the cloud. The first algorithm is a device-oriented algorithm, in which each devices are independent and proactively optimize its recognition rate by balance the exploitation and exploration of different levels of network contention and cloud congestion. While the second algorithm is service-oriented and it takes into account the resource restriction of the cloud system and allow the cloud to learn the best transmission setting and suggested to all the devices for efficient resource usage.

## 2.5 Malware Information Sharing Platforms

Collaborative threat analysis and threat information sharing gain more and more popularities in mitigating complex cyber attacks, but the actions are mostly restricted inside the organization's ecosystem. Webroot's BrightCloud platform is an exemplar collective threat intelligence system that apture, analyze, classify, correlate and publish cyber threat intelligence. In the tech brief [2], it developed an overarching view of the thread landscape from integration of billions of pieces of information from millions of security sensors. For example, the collective information showed 85,000 malicious URL launched daily. Among the 40 million new files saw by its clients network in 2014, 15% are malware. We see only through collective information can we know the malware landscape in this level. However, the BrightCloud didn't have a model that are automatically and accurately conduct the analysis and classification.

The Verizon 2015 Data Breach Investigation Report [37] shared some status about threat intelligence sharing. In short, two facts are shared: 1) Threat feeds on sources of scanning activity and spam/phishing e-mail are significantly overlapped, while threat feeds that provide information on destinations that serve either exploit kits or malware binaries, or provide locations of command-and-control servers are barely overlapped. 2) 75% of attacks spread from one to another within one day, and most of those indicator remain valid only for 1 days. The report also present study from ThreatConnect what exactly shared among its sharing community. In the Figure 7 of the report [37], IP addresses and hosts are highly shared(around 50%); however, few E-mail, files, URLs are shared among the studied communities(less than 10%). This could be our motivation to increase file sharing, because files might be the most complex form of data to share, analyze, and detect.

Security Information and Event Management (SIEM) suite is one of the most broadly deployed threat sharing system among organizations that have established mutual trusts. Most SIEM systems mainly rely on human analysts to first identify threat (i.e., phishing urls, indicator of DDos, targeted malware campaign, etc.) by forensic analysis or reverse engineering. The intrusion detection system achieved better performance by simply accumulating the manually identified threat information which essentially demand significant amount of human labor.

Recently, the sharing initiatives from organizations including government agencies, companies, and individuals have shown a willingness to share threat intelligence across organizations. Malware Information Sharing Platform [38], designed by NATO, promoted the concept of cyber defense information sharing. It is a open source project with a combination of a community of members, a knowledge base on malware, and a web based platform. STIX, TAXII and CybOX specifications [39] and MAEC language [40] by MITRE Corporation is a series of the most prominent work attempting to standardize the vocabulary, expression and conveyance of threat intelligence thus to solve the challenge that data could not be automatically and effectively transfered and consumed. With these specifications from MITRE, the sharing processes could be done automatically across heterogeneous networks and even between disparate tools. Many industry platforms adopted STIX, TAXII, and CybOX to deploy on-line threat sharing communities that is accessible to public. Representative platforms we reviewed include AlienVault Open Threat Exchange (OTX) [41], ThreatConnect collaborative threat intelligence platform [42], IBM X-Force Exchange [43], Facebook ThreatExchange [44], Cyber Threat Alliance [45], and Virus Total [18]. These platforms is capable of sharing thread information in a social network fashion, in the format of a list of malicious IP address, malware analysis reports, indicator of compromise, or malicious executables.

We've saw the model of sharing threat information is desirable and highly practical. Participants recognize that collaborative threat sharing is beneficial. However, the general idea of threat sharing outside organization is unfavorable to those who argue that there is a chance of losing privacy or losing competition advantages. To better understand this threat information sharing paradigm, we build a centralized system which accept public contributed threat data and run on it various analysis algorithms to achieve automatic intrusion detection; More importantly, we model the system operation from a public goods game theoretical point of view, from which we could gain insights about the conflict between individual incentives to free-ride and social incentives to contribute toward the provision of public intrusion detection system. Our experiment will focus on malicious file sharing and collaborative malware analysis. To our best knowledge, there is no existing work on theoretical modeling of a practical threat information sharing system.

CHAPTER 3

# TroGuard as a Host-Based Malware Detector

## 3.1 Background

Social engineering attacks rely on the user being a point of weakness in any secure system and create one of the most challenging security problems, where users themselves perform or facilitate attack steps. For example, in the case of "fake anti-virus" malware, the user is convinced to download the legitimate-looking executable to her computer and then willingly execute a piece of software that is core to the attack. Protecting a computer system and its users against such socially engineered download attacks appears to be impossible, as any such protection could interfere with the user's freedom to install software on their own computer. We describe in this paper our efforts to alleviate this threat without impinging on the user's freedom, by providing a system that automatically compares the stated goal of a software program (as perceived by the user) with its actual goal (as inferred from the program's execution).

Web-based socially engineered download attacks that result in a trojan software installation on the victim's computer are becoming more widespread. This is mainly due to new security measures (e.g., memory-page protection and address randomization to prevent buffer overflows) that significantly reduce the success rate of other popular, automated attack vectors. Trojan downloads range from rogue security software (also known as "scareware" or fake anti-virus), to fake games, fake video codecs, and to fully functional pirated softwares that are infected with malware. As a case in point, `TrojanClicker.VB.395` (detected by TrojanHunter [46]) purports to be an Adobe Flash updater; however, once downloaded and installed by unsuspecting users, the trojan launches a spyware that monitored and uploaded all Google searches to a remote server `www.msjupdat e.com`. To make the users aware of such attacks, Adobe announced that malicious hackers were starting to use fake Flash Player downloads as social engineering lures for malware and issued a call-to-arms for users to validate installers before downloading software updates. The company's notice came on the heels of malware attacks on Facebook, MySpace and Twitter that attempted to trick Windows users into installing a Flash Player update [47]. Furthermore in the last few years the threats of trojan downloads have expanded quickly from desktop to mobile platforms, e.g., the Opfake browser [48], where the app-store concept seems to drive the user-made download decisions away from provenance and towards functionality. t There are unfortunately few techniques to help users protect themselves against trojan downloads. One technique is based on dynamically updated blacklists, where web browsers check against an online blacklist service (e.g., Google SafeBrowsing [49]) that the current web page does not contain malicious software. Similarly, anti-virus software (e.g., McAfee [50]) also uses a blacklist to determine whether the downloaded program is malicious. Both of these security techniques rely on precise and timely maintenance of the blacklists—a time-consuming challenging endeavor in practice. Protection techniques that monitor for "drive-by downloads" (e.g., Blade [51]) are inapplicable here because trojan downloads are performed through user interaction, not via an exploit. User education to protect against trojans is ineffective at best since the vast majority of users lack the necessarily complex technical skills and tools to determine whether a downloaded program is malicious. Consequently, almost all of the existing techniques fall short in providing an effective protection mechanism against trojan download attacks that could ideally manage the whole download-install-execute process for the user and provide security in that context.

Our approach bridges the gap between what the user perceives regarding the functionality of the downloaded software and the functionality observed during its actual execution. We observe that, throughout the download–install–execute process, the user maintains a consistent expectation about the functionality of the downloaded program. Following the Principle of Least Surprise [52] in user-interface design, i.e., "design should match the user's experience, expectations, and mental models" [53], we wish to build on the user's expectations of a downloaded application and her experience with past applications with similar functionalities. To this end, our solution detects trojan downloads by comparing the user's expectation of functionality (as primed by the software download web page) with the actual functionality exhibited at runtime, and imposes constraints on the downloaded application's local execution using the contextual information on the download web page.

Comparing an application's expected and actual behaviors, while intuitive, poses significant challenges. It is unclear whether an actual system-level behavior can be summarized concisely that is high-level enough to compare with a label that a typical human user could place on the downloaded application. This is mainly because of the huge gap between the human description of the application (in terms of abstract concepts and functionalities such as web browsing) and its system-level behavior (in terms of concrete operations and concrete runtime metrics such as syscalls). Additionally even obtaining a concise label from the human description is not straightforward, as web pages for application downloads often contain extraneous marketing information. We tackle these challenges to create an end-to-end system for detecting trojan downloads.

Our system, called TroGuard, analyzes the information visible to the user on the software download web page to infer the perceived software category. TroGuard also monitors the execution of the downloaded program to infer its actual software category in real-time, and alerts the user if the perceived and actual categories do not match. We chose to abstract as *software category* both the web-page information around the program download and the execution of the program, because a software category matches closely to how users commonly understand software. Finding this middle ground between human descriptions of software functionality and low-level system operations invoked by the software is non-trivial, because those software categories need to be not only meaningful to users (e.g., “office application” vs. “text editing tool”) but also extractable from the low-level system traces. Furthermore, applications within the same category should expose similar system-level profiles to be classified as the same category by TroGuard later.

Even with a well-chosen set of software categories (as we give in this paper), there are significant engineering problems in (1) designing a web-page analyzer that can automatically classify the download with high accuracy, in the presence of rich web technologies fully under the attacker’s control, (2) constructing a runtime monitor that can abstract away inconsequential behaviors so to classify the execution of the downloaded program correctly, and (3) integration of the complete TroGuard framework into a convenient web-browsing experience for the user. As detailed throughout the paper, TroGuard overcomes these problems through a combination of machine learning and security engineering.

We use supervised machine-learning to construct our software-category classifier. We considered two training data sets, one consisting of traces collected from user-executed software and one from symbolically-executed software. Although the first data set does not provide complete coverage, since our users did not attempt to explore all functionality of each software program, it leads to a more accurate classifier. We explore and explain this surprising phenomenon in our evaluation.

We make the following contributions in this paper:

- A new approach to detecting trojan downloads is introduced, that at its core relies on comparing the functionality expected by the user (efficiently determined through in-depth and automated analysis of the download web site) to the functionality exhibited during the downloaded application's execution;

- An end-to-end system for detecting trojan downloads is designed and implemented to identify mismatches between user-perceived and actual software categories. TroGuard consists of a browser extension and plugin as well as a host monitor that communicate with each other to bridge the gap between user experience and system activity; and

- A comprehensive evaluation over a large data set shows high-accuracy detection (up to 98.3%), even in the presence of programs that naturally combine functionality from multiple (traditional) software categories.

It is also noteworthy to mention that TroGuard concentrates on the *detection* of socially-engineered trojans and does not provide post-detection intrusion-response solutions to remove the malware or restore the system back to a previous clean state.

Table 3.1: 90% of the top MacOS Malware are Trojans [1].

| **Rank** | **Name** | **Percentage** |
|---|---|---|
| 1 | Trojan.OSX.FakeCo.a | 52% |
| 2 | Trojan-Downloader.OSX.Jahlav.d | 8% |
| 3 | Trojan-Downloader.OSX.Flashfake.ai | 7% |
| 4 | Trojan-Downloader.OSX.FavDonw.c | 5% |
| 5 | Trojan-Downloader.OSX.FavDonw.a | 2% |
| 6 | Trojan-Downloader.OSX.Flashfake.ab | 2% |
| 7 | Trojan-FakeAV.OSX.Defma.gen | 2% |
| 8 | Trojan-FakeAV.OSX.Defma.f | 1% |
| 9 | Exploit.OSX.Smid.b | 1% |
| 10 | Trojan-Downloader.OSX.Flashfake.af | 1% |

### 3.1.1 Motivation

The attacks via web-based Trojan downloads are hard to protect against because they involve a social engineering step where the user (as sociotechnical root of trust) gets tricked (i.e., compromised) into downloading and running the Trojan program themselves. This favors attackers such that nine out of the top-10 MacOS threats in 2012 were Trojan downloads (Table 3.1) causing 30% increase in number of signatures created by Kaspersky Labs [1].

The ubuntu malware [54] hidden inside a screen saver from Gnome-look.org was one of the real word examples showing the severity of such attach in non-Windows platform. When a ordinary computer user downloaded this screen saver without knowing anything about the downloaded executable, the installed application secretly install some executable scripts beside giving the user the desired screen saver functionality. The secretly installed scripts connected to command and control server for downloading malicious code or steal local sensitive information.

We tackle this problem by equipping the user with a tool to answer the question “Is this downloaded program doing what I expect it to do?”. To this end, we combine an analysis of the information about the program that is available to the user on the download web page with an analysis of the runtime information collected by the user's computer from the program execution. Both of these analyses produce summaries of the program functionality, one of perceived functionality (as the user sees it) and one of actual functionality (as the host system sees it). The perceived and actual functionality profiles are then compared against each other and the user is notified in the case of mismatch. The analyses rely on a database of known applications, known websites, and known functionality classes in order to produce the perceived and actual summaries.

## 3.2 System Architecture

We present an overview of the proposed solution and discuss how it protects the client systems against web-based socially engineered Trojan threats. Figure 3.1 illustrates individual components in TroGuard and how they are logically interconnected. The overall solution consists of two major steps of 1) offline application tracing and data analysis, and 2) online user query and dynamic malware detection.

More specifically, during the offline phase, TroGuard captures and traces several behavioral aspects of various applications with different *functionalities* dynamically. TroGuard analyzes the program from a collection of labeled applications where each application, e.g., Firefox, is marked with the functionality it provides, e.g., web browser. The abovementioned collection is created such that most of widely-used functionalities, e.g., web browser or office suite, are provided by multiple applications. The objective of the offline phase is to create a behavioral profile of each functionality in terms of its system-level activities, such as filesystem reads/writes, network data transfer, and user interactivity. The premise, that we prove by our experimental eval-

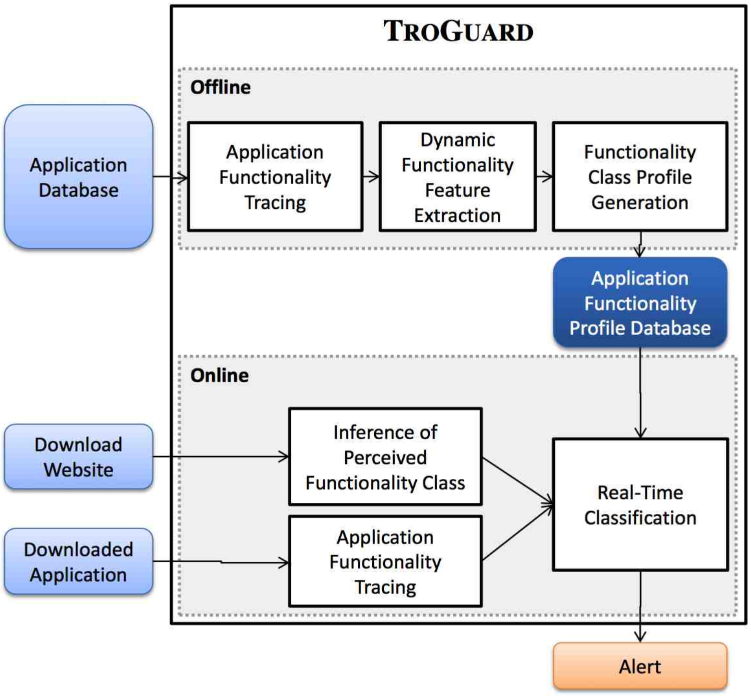

TroGuard
Offline
Application Database
Application Functionality Tracing
Dynamic Functionality Feature Extraction
Functionality Class Profile Generation
Application Functionality Profile Database
Online
Download Website
Downloaded Application
Inference of Perceived Functionality Class
Application Functionality Tracing
Real-Time Classification
Alert

Figure 3.1: TroGuard's High-Level Architecture

uation, is that behavioral aspects of applications with similar functionality are similar to each other and different from those of applications with different functionalities.

Once the application activities are captured, the recorded data sets with the functionality labels are fed into a supervised machine-learning algorithm to extract discriminative features about each functionality. For instance, game applications often exhibit intense user interactivity and time-varying system CPU usage while music players present minimal user interactivity and stable CPU usage over time, with low filesystem and network interaction. Similarly, word-processor users will press navigation and alphanumeric keys significantly more than other key categories, while for game players either use a limited times of alphanumeric keys such as A, S, D, and W. The discriminative features for each functionality form a profile feature vector and are stored for the online detection phase. It is noteworthy that the offline phase is a one-time effort and the functionality profiles, once created, can be reused across multiple systems.

The second phase in TroGuard is that of on-demand website and application analysis, leading to the automated real-time malware detection. When the user intends to download a legitimate-looking (potentially malicious) program, e.g., a game, from a website, TroGuard monitors the download and installation process on the local system closely using its browser extension and plugin components, respectively. Once the browser extension component in TroGuard notices a download initiated by the user, it starts the website analysis process to determine the type of application (functionality) being downloaded, by parsing the text from the web site and from its images (which are first processed using OCR). TroGuard then calculates an list of most-likely functionalities and presents them to the user before the download starts. The user confirms or corrects the expected functionality for TroGuard to use during its dynamic program analysis. Once the program is downloaded, installed, and executed in a sandboxed environment (all steps automatically monitored), TroGuard's runtime tracing component starts behavioral monitoring, data collection, and real-time classification of its behavior to verify that the application does not violate the corresponding functionality profile obtained from the offline phase.

Revisiting our example of `TrojanClicker.VB.395` from the introduction, TroGuard would infer that the program to be downloaded claims to be a software updater and would retrieve the appropriate functionality profile from its database. When the Trojan starts executing, TroGuard would monitor its activity, looking for updater-like behaviors (e.g., many filesystem and registry operations for existing entries, little-to-no network traffic). As the Trojan installs itself into the system (into a new location and as an extension to any web browsers present) and as it communicates over the network (sending collected logs of Google searches), TroGuard observes the discrepancy and notifies the user consequently.

TroGuard protects the users against a specific class of Trojan download attacks where, through a web-based social engineering step, the user is lured into downloading and installing a malicious legitimate-looking executable on her system. A web-based Trojan attack download can be described as a series of steps that the victim is initially tricked using social engineering, and then he or she intentionally performs to complete the download and installation of malware for the attacker. The goal of the web-based Trojan exploit is to take effective control of the client machine in order to complete subsequent malicious activities such as bot deployment.

The attacker wishes to trick the user into downloading the Trojan program, and can use any social engineering technique for this purpose. Additionally, the attacker has full control over the web page that proposes the download to the user and can use any web technology on that page, with the exception of exploits. In other words, we place the user's web browser and underlying operating system into the trusted computing base (TCB) and assume that they have no vulnerabilities. Protecting the TCB from attacks is orthogonal to our work and has been widely researched by others, with a variety of solutions available.

We assume that the attack does not involve the use of exploits or other automated techniques, but only the willing cooperation of the user to install and launch the Trojan software. It is important to highlight that our threat model is subtly different from drive-by downloads (e.g., as addressed by Blade [51]), where the adversary performs a *surreptitious* download via the user's browser through a shell-code injection step, with no social engineering needed. In this case a browser vulnerability is exploited and no social engineering step is needed to complete the attack. Furthermore, unlike in our threat model, the user does not notice the download and installation of the malware. This type of drive-by download attack requires a vulnerable browser to be feasible and is outside the scope of the present work.

TroGuard analyses the source website of an application, while that application is being downloaded, to determine the appropriate functionality profile for that application. It then compares the execution of the downloaded application against this profile and alerts the user if it discovers any discrepancies. In this section, we discuss what a functionality profile is and how we build a representative database of them.

### 3.2.1 Functionality Classes

*Functionality class* is a key concept in our system, as it represents both the user's understanding of a software category and the system's observation of a software's execution behavior. Defining an ideal structure to capture the concept of functionality class is close to impossible, as an unlimited number of structures of varying degrees of abstraction could serve the purpose.

In TroGuard we use a highly summarized view of the program execution to define a functionality class. In particular, the profile associated with a functionality class is a vector of key behavior features, as explained in the following subsections. Each behavior feature is high-level enough to be describable to the user (e.g., "Program X was classified as a game because in normal use it generates a lot of user interaction", where *level of user interaction* is one feature in the profile vector). Further more, each behavior feature should be efficiently computable in real time from the program execution observations.

The remaining challenge is how to select the most representative functionality classes. By "most representative" we mean both classes that are meaningful to users

| **TroGuard** | **Softpedia.com** | **cnet.com** | **Tucows.com** |
|---|---|---|---|
| Graphics Editor | Artistic SW | Graphic Design SW | Design tools |
| Game | Games | Games | Games |
| Browser | Internet | Browsers | Internet |
| Instant Messenger (IM) | Communications | Communications | |
| Media Player | Multimedia | MP3/Audio SW | Audio / Video |
| Audio Editor | | | |
| Video Editor | | Video SW | |
| Office | Office | Productivity SW | Business |
| IDE | Programming | Developer Tools | Dev/Web |
| Calculator | Utilities | Utilities/OS | Home /Education |

Table 3.2: Functionality classes in TroGuard and their corresponding categories for three popular software-download web sites.

and with member programs that share behaviors. As an example of a poor choice of functionality class, "System Utilities" fails our criteria because its meaning is too generic to a user and its member programs have many, distinct behaviors with no common functionality. We turned to ten software-download websites such as Softpedia, Tucows, and FileGuru, and surveyed their top-level software categories. As these website are popular Internet destinations for software downloads, their categories likely reflect users' understanding and serve as a good starting point for TroGuard. We summarized the categories and combined them into a more concise set, as shown in Table 3.2.

It is noteworthy that the correct selection of functionality classes results in effective detection of behavioral mimicry attacks, where the carefully crafted malware attempts to accomplish a malicious objective while pretending a legitimate functionality delivery. We will discuss how robust TroGuard is against mimicry attacks in section 3.4.2.

### 3.2.2 Bridging the Semantic Gap

The goal of our system is to distinguish an application's genuine type from its advertised type based on functional profiling of the application against a trained application functionality class database. The system should first be trained using a supervised training algorithm with a large set of sample applications. To generate discriminative features, comprehensive functional profiles are required. Unlike previous work on behavior based malware clustering [55] and [56], besides system call traces, application behavioral in our work is expanded to include user space properties: CPU and memory usage, network protocols, port numbers, the number of IP addresses (or domain names) the application connects to, and the user machine interactivity, such as keyboard strokes and mouse clicks. The presence or absence of those attributes in each application's profile not only represent the core functionalities of that application, but can also be used to separate a particular type of application from the rest. Practically, such behavioral profiles are generated by collecting and analyzing the system wide activities both in kernel space and user space. In this section we will discuss the feature list generation and it properties.

System call traces are very useful in studying the run-time activities of an application. In our project we use the LTTng [57] tool kit to obtain the system call logs. However, we will not analyze the system call log directly, such as using the n-gram method. This is mainly because system call traces can vary significantly between programs of a single application type. Consider a file downloader as an example: in a multi-process platform, file downloader A might write 512 bytes data fetched from the web server to the file system in eight consecutive write system calls, taking 64 bytes as the input argument for each write operation. Whereas, file downloader B might write 512 bytes to the file system by a single write operation. Moreover, in the trace of program A, it is very likely to interleave the write system calls with some other process management system calls. The significant differences in such system call traces of two similar applications will not reflect the similar behaviors they exhibit. For this reason, we first study the system call log manually and build a model to abstract feature from it.

TroGuard extracts low-level system features using its dynamic kernel and user space tracing engines. For low-level kernel activity tracing, TroGuard intercepts syscalls, and it analyzes system call to obtain more semantic information such as used protocols, system libraries that the application is accessing, file system activities. Furthermore, we focus on the input and output arguments of the system call as well as the system call statistics such as average amount of network data transfer. In order to obtain the dependency of the data flow from the trace, we also consider the functionality of individual system calls. This is necessary to build the dependency relationship among system entities. Additionally, TroGuard gathers system call statistics such as system call frequency in different operation categories, i.e., file system I/O statistics, and network I/O statistics, filesystem accesses, and interprocess communication. Though system call analysis, TroGuard also extracts the information flow dependency among the operating system assets, e.g., processes and directories. For instance, when a process sends an IPC message to another running process, the second process becomes directly (information flow-) dependent on the first process. Transitive information flow among a sequence of assets cause indirect dependencies. TroGuard adds such data dependencies to the system-level features of the target application. Based on our system call trace analysis model, a list of feature vectors are generated from the kernel space trace, including the total number of system calls in different operation categories, file system I/O statistics, and network I/O statistics, file modification dependency, and interprocess communication.

In addition to the system call traces, we include some user space properties as part of a program's feature list. These include CPU and memory usage, user machine interactivity, port numbers, and the number of IP addresses (or domain names) the application connected to. It is worth mentioning that some of those properties are theoretically available in the kernel traces. However, due to the limitation of our tracing tool, the port numbers a program used could not be retrieved from the traces and we collect them using Linux utility programs such as `sockstat` and `lsof` out of the kernel space. The whole feature collection process is done by running a script, in which both the LTTng tracing results and the `sockstat` or `lsof` output are collected as raw data files. Those files are further processed and the output of this step is the abstracted application profile.

We consider CPU and Memory usage as application features because during our study we found that CPU and memory usage for different types of applications exhibit similar patterns that are unique to that application type. For example, the resource consumptions for web browsers have a sharp increase at start up, and are then reduced to a moderate level and linearly increase as new tabs are created. While for text editors, the CPU and memory usage mostly remain in the same level as it started. Moreover, resource consumptions implicitly reflect how the user interacts with the application. Resource consumptions such as CPU and memory usage are correlated with the tasks the program executes. The human interactivity features are also considered in our feature list in the form of statistics of keyboard strokes and mouse clicks.

Nowadays, most applications highly rely on a remote web server. The port numbers and IP addresses the program communicate with can be taken as features in the feature list. For example, port 25 is for SMTP service, so any application that uses port 25 is very likely to be an email client. By dumping the IP addresses and referring to the open website directory such as dmoz.com. we can link the program to a particular category of web services, which is used as one element in the feature list for classification later.

The main challenge in designing such functionality-based detection solutions is the large semantic gap that exists between the high-level user-perceivable functionality classes (Section 3.2.1) and low-level system activities, e.g., syscalls. To fill the semantic gap without sacrificing detection accuracy much, TroGuard uses an intermediate level of features that not only are semantically closer to the functionality classes but also can be inferred from the low-level system and network event logs (Figure 3.2). In particular, the system-wide tracing instrumentations notifies TroGuard about low-level activities. TroGuard accomplishes intermediate-level feature inference from the low-level activities using predefined rules implemented by regular expressions. The inference rules are designed as a one-time effort (took two person-days in our experiments) and, once developed, could be reused across machines. The inference rules are either direct or indirect. TroGuard directly infers an intermediate-level feature when it explicitly matches particular points in the system-level activity logs such as a file type access attempt or a specific network protocol. Indirect inference requires more computation since the information should be extracted and combined from different points such as system call arguments or system call invocation sequences, and processed to check whether they satisfy a given regular expression rule. For instance, the intermediate attributes `protocol::imap` and `works-with-format::pdf`, belong

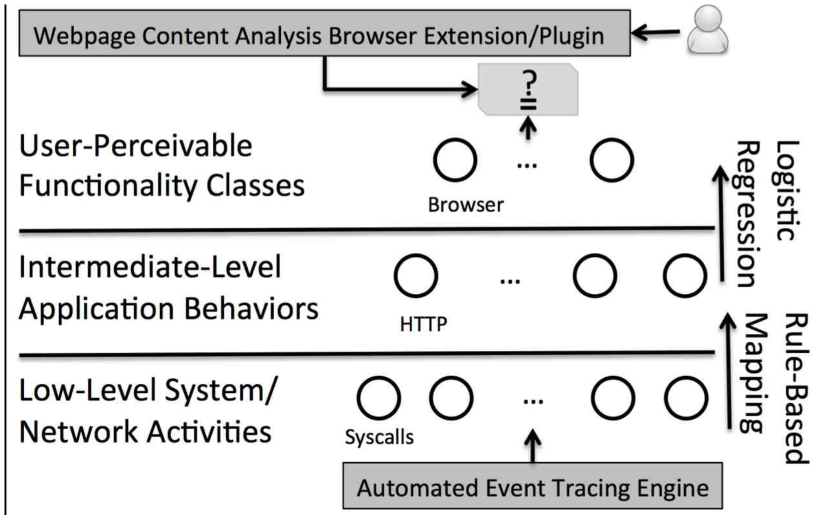


Figure 3.2: TroGuard Components

to the directly mapped attributes extracted from Debtags system [58], can be obtained explicitly from system call argument. However, the feature `interface::x11` is inferred indirectly from low level system call activities such as opening and frequently reading Xorg library files.

We take Debtags information from Debian package manage system as a reference in writing the intermediate feature inference rules. We believe that the high level application functionality classes of those packages can be identified by combination of multiple tags from Debtags [58]. In other words, a subset of Debtags can play the role of feature vector to distinguish applications in various categories. Furthermore, if this particular subset can be generalized to classify other applications except Debian packages, it will be the feature vector we are looking for. To verify our hypothesis, we need to collect tags for our studied applications coming from outside of Debian package repository. In our implementation, we map system level features such as system call traces to intermediate level features (Debian tags) by rules determined empirically.

### 3.2.3 Completeness of Functionality Profiles

The functionality profiles we construct in TroGuard are based on the application behaviors observed during a training phase, in which selected, well-known applications are run manually under comprehensive system monitoring. This necessarily means that the functionality profiles are incomplete, which could lead to false alarms (e.g., alerting that a downloaded copy of the Banshee music player is not really a music player).

In our current implementation we considered two options for the training phase. The first option relies on manual exploration, where users exercise most of the core functionality of each application in the training set. While this does not technically ensure 100% coverage of code paths, it sufficiently captures the code paths commonly executed by normal users. The second option relies on symbolic execution [59] to improve the code coverage and thus the functionality coverage. We will explore this option in future work. The full results will be delivered in this thesis.

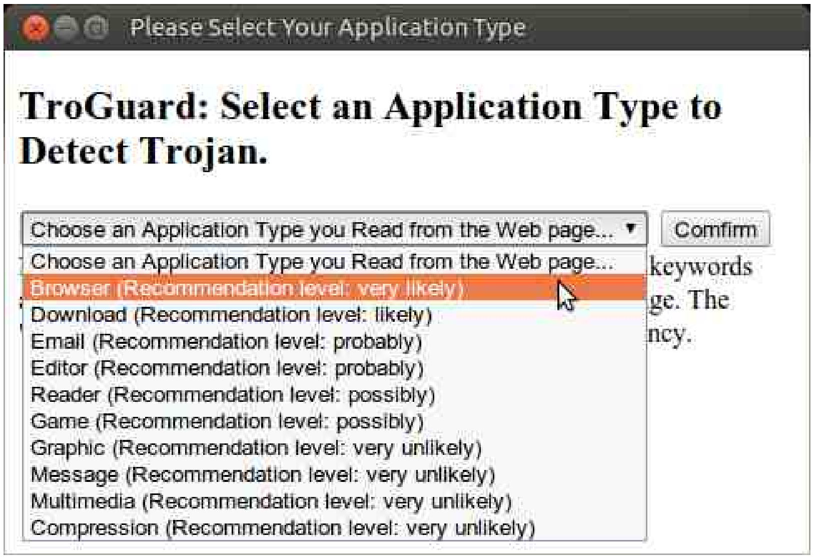
## 3.3 Implementation

### 3.3.1 Browser Extension

The webpage analysis is implemented as an extension to the Google Chrome browser. The real-time detection system composed of Chrome extension with web page contents analysis, application tracing component, and NPAPI plugin (also implemented in C++) that can be used by the extension as interface to the local host to initiate the detection engine. The extension monitors the user's browsing behavior and whenever a file download link is clicked, the extension runs the web page analysis (implemented in JavaScript) and presents the user with a dialog box before the download starts. The user then uses the drop-down selection (Figure 3.3) with the most likely application type ranked as its first option. The likelihood ranking is computed by the web page analysis component of the extension. The user has the opportunity to select another application type (according to his expectation). Once the perceived functionality type is confirmed, the extension invokes (via a native NPAPI plugin component) the following tasks in our system: initiation of system-call tracing, initiation of user-space feature collection, and application classification. By comparing with the functionality class selected by the user, the detection system can warn as appropriate.

TroGuard fires up when the user interacts with the web browser to download a software application. When the user clicks a file URL to download an application in a TroGuard protected computer, the browser extension invokes a popup window with drop-down options displaying a set of potential application types extracted from the web page. The displayed application types are a set of frequently occurring keywords such as web browser, email client, and text editor, etc. These keywords are defined during the design and implementation of the browser extension. When the user confirms the application type by either confirming the default type or selecting another type she/he believes the program belongs to, this keyword is submitted to the extension and used later for the detection step.

TroGuard uses the Chrome Extension technology [60], to bring up a popup window (Figure 3.3) which provides the user with a list of potential application type he/she is currently downloading. TroGuard performs website content analysis in the background prior to showing the list to the user. The content analysis component of TroGuard makes use of the text in introducing and marketing the product, as well as related pictures on the current web page. This component automatically



Please Select Your Application Type

**TroGuard: Select an Application Type to Detect Trojan.**

Choose an Application Type you Read from the Web page... ▼ | Confirm

- Choose an Application Type you Read from the Web page...
- Browser (Recommendation level: very likely)
- Download (Recommendation level: likely)
- Email (Recommendation level: probably)
- Editor (Recommendation level: probably)
- Reader (Recommendation level: possibly)
- Game (Recommendation level: possibly)
- Graphic (Recommendation level: very unlikely)
- Message (Recommendation level: very unlikely)
- Multimedia (Recommendation level: very unlikely)
- Compression (Recommendation level: very unlikely)

keywords
ge. The
ncy.

Figure 3.3: Proof of concept example popup window

search all the DOM nodes in the current web page, captures screen-shots of the visible surface of the browsing window, and employs an OCR engine to extract the text in the screen-shots. We use Tesseract [61] as the OCR engine. Specifically, we study the websites front end content to obtain software vendor's favorite word choices in describing their applications. Some application names are described in a single keywords, such as browser and game, and those keywords are descriptive of the application type on their own. If the content analysis component extracts any of these descriptive keywords from the download page, the corresponding application type is added to the list of potential applications shown to the user. For other less descriptive application names, TroGuard requires co-occurrence of multiple keywords on the page to characterize its application type in website analysis. For instance, we count the appearance of both "message" and "chat" in the web page to add Instance Message application type to the potential application list.

The following is a list of applications and their descriptive single keywords: Browser (browser), Game (game), IDE (develop), Media Player (player), Calculator (calculator), Downloader (download), Email (email), Database (database), Driver (driver), Ebook (ebook), and Themes (themes). The second category of applications without a single descriptive keyword include Office (text, editor), IM (chat, message), Graphics (image, draw), Video Editor (video, editor), Audio Editor (audio, editor), PDF Reader (pdf, reader), Anti-virus (security, antivirus, virus), P2P Application (share, download, p2p), Network Tools (network, ip, internet). The application types in the drop-down menu are ranked according to the frequency of those keywords appearing in the web page, with the highest ranking type shown as the default option.

Furthermore, to simplify TroGuard's operations on the user's side, we have created a simple PHP web service that receives the screen-shot URL, extracts the text using a PHP wrapper around Tesseract and returns the extracted text. TroGuard's Chrome Extension talks with the PHP web service via simple AJAX requests. The application types in the drop-down menu are ranked according to the frequency of those keywords appearing in the web page, with the highest ranking type as the default option. Specifically, empirically define When the user confirms the application type by either confirming the default type or selecting another type she/he believes the program belongs to, this keyword is submitted to the extension and used later for the detection step.

### 3.3.2 Application Tracing

The runtime tracing is started as soon as the downloaded executable is launched. The tracing operates by monitoring low-level system activities, mapping them to intermediate-level application behaviors, and then checking these intermediate-level behaviors against the functionality profile named by the functionality class inferred from the download web page. If the intermediate-level behaviors do not match the functionality profile, an alert is raised and the user is given the option to terminate the application.

The captured low-level system features consist of intercepted system calls, CPU utilization, memory usage, keyboard activity, mouse clicks, and network activity, e.g., socket numbers and IP addresses.

TroGuard further monitors the child and peer processes, and assigns their OS-object accesses to the top application. Peer processes are processes that were influenced by the application or one of its peer processes via inter-process communication (IPC), shared memory, remote-process calls (RPC), or other cross-process mechanisms. Consequently, TroGuard adds the assets that interact with the application directly or indirectly to the target application's feature vector. This aggressive approach to assigning all direct and indirect resource accesses to the top application ensures that all malicious actions are visible to TroGuard.

Additionally, TroGuard also takes into account the frequency and amount of transferred data among the assets as well as between the local host and remote network end-points. For the application's network activity, TroGuard also records the average number of unique remote end-points that the application communicates with every time unit. To capture the user-machine's interactivity, we calculate statistical measures to encode the degree and type of interactivity from observed input events. In particular, we divide keyboard keys into several types: alphanumeric, modifiers, navigation and typing modes, system commands and miscellaneous [62]. For mouse clicks, TroGuard captures left/right/middle clicks and scroll wheel rotation.

TroGuard is primarily a monitoring, classification, and detection framework and as such has to be supplemented with mechanisms for protecting the host environment from the untrusted application's execution. Sandboxing is the best choice of mechanism in our scenario, as it cleanly separates the application's persistent changes to the OS from the rest of the host system. TroGuard makes use of SE-Linux platform to generate and enforce policies to keep the downloaded applications within a contained sandbox until their benign behavior is confirmed by TroGuard. During the training phase, TroGuard logs the executed syscalls for the applications in a functionality class and then uses the union of those logs to generate SE-Linux policies automatically. TroGuard creates a policy ruleset by parsing the log files for each functionality class. Once the user downloads an application, TroGuard employs the created sandbox (i.e., loads the corresponding generated policy's kernel modules) for the claimed functionality class. Consequently, the loaded sandbox policies will not allow the downloaded applications to expose behaviors not already exposed by any of the application instances within the same functionality class during the training.

A variety of sandboxing approaches work in combination with TroGuard, ranging from whole-system virtualization [63], to per-app virtualization [64], to emulation [65], and to microvirtualization [66]. Since TroGuard is not a static-verification to ol, but relies on dynamic analysis, it can never safely declare an application as non-malicious (especially given the existence of time-delayed trojans [67]). As such, a downloaded application has to be sandboxed in every run. We note however that, TroGuard allows us to merge persistent changes from the sandbox into the host system, because TroGuard reasons only about a recent window of events from the application. All past activity from the application can safely be committed to the host system as long as TroGuard did not raise an alarm about them.

### 3.3.3 Monitor for Execution Analysis

The native components, outside the browser, consist of a system-call tracing tool and a user-space component for collecting high-level information, both for Linux.

The LTTng tool kit is used in our project to collect system call traces. LTTng project aims at providing highly efficient tracing tools for Linux platform. It has been frequently used for tracking down performance issues and debugging problems involving multiple concurrent process and threads. LTTng kernel tracer is used in our project and is installed as a Linux kernel module. The tracing result written to file system in a format called Common Trace Format (CTF) is then processed using Babeltrace (part of LTTng) to generate a log that can then be analyzed.

The profiler functions as trace parser and data formating tool (implemented in C++). It is not only to parse and extract file system access information but also to analyze informative returned value, and resolve the interdependency between the different system calls. For example, it analyzes the possible system call patterns between `open` and `close` and the `path` arguments of the `read` and `write` operations to build file system access dependencies. The parser takes care of all the file system modification operations in newest kernel versions. It also parses and calculates the data amount the particular application process writes or reads, and the amount of network data it sends and receives. We resolve the process dependencies by the following heuristic. We say that process $B$ depends on process $A$ if process $B$ reads a file $F$ after process $A$ wrote to file $F$.

The user-space feature collection for CPU and memory usage, user machine interactivity, port numbers, and the total numbers of IP addresses relies on Linux utilities such as `sockstat` and `lsof`. The combination of kernel space system call traces and user space meta-features enable us to generate comprehensive application profiles. In our implementation, the CPU and memory usage are obtained by dynamically reading the `/proc` file system data structures. The user interactivity feature consists of statistics over keyboard strokes and mouse clicks. We divide computer keys into several types: alphanumeric, modifiers, navigation and typing modes, system commands and miscellaneous [62]. For mouse clicks, we track the following attributes, left click, right click, middle click, and scroll wheel activity. We use `evtest`, a input device event monitor and query tool in the X11 server platform, to capture all the keyboard strokes and the mouse clicks. The results are presented as XML files and read by our profiling program later to generate the application profiles.

### 3.3.4 Functionality Profile Generation

All of those applications were installed incrementally and run separately in a Ubuntu virtual machine with LTTng module loaded. An automatic data collection bash script was used to collect kernel trace result by running LTTng and initiating other tools to collect resource usage information, such as CPU and memory usage. During each application tracing, our researcher worked with the application to explore its features as most as he can for 60 seconds. The output of the raw data collection is one large size system call log and five small size feature files, including the CPU usage, memory usage, keyboard logging, mouse click logging, and the dumped network information (socket numbers and IP addresses). The profiler will take those files at the collection step as the input to produce a single line of entry in the final dataset, which in our case is single ARFF file for all the 100 applications. In details, the CPU and memory usage is read from `/proc/[pid]/stat`. The socket and IP information is obtained by running `netstat`. The I/O devices is monitored by `evtest`. By doing preliminary test and statistical analysis of keyboard key usage for variance applications, we observed the pattern of usage for the keyboard key between different type of applications.

We employed machine learning tool Weka as our training and classification engine. The application profiles generated as Attribute-Relation File Format (ARFF) format in order to meet the supported input format for Weka. ARFF is one of the simple data set format Weka can parse correctly.

We selected 100 applications in 10 categories from Ubuntu Software Center and Softpedia.com. To build the application profile database, the testing platform has been configured to allow file sharing between guest and host. So the application traced results can be immediately profiled by the powerful host machine. Which can significantly reduce the time to build the application functionality profile database.

Total number of 100 applications have been installed and the tracing results have been obtained inside the virtual machine. To reduce the system call trace data size and accelerate the profiling process, we ensured that there are no other user application run except the target application and the system service processes the operating system is needed in bootstrap. In the off-line training stage, each of the application was run for 60 seconds with user performing normal operations on the application main functionalities. After obtain the raw data, which will be pass to the trace parser for generating application profile. Each application profile collected during 60 seconds is divide into 6 independent data point with 10 second trace for each. We put together the whole data set as a single ARFF data file after all the traces have been completed.

## 3.4 Evaluation

We deployed TroGuard in a testbed environment and evaluated various aspects of its operation. In particular, we designed a set of experiments to empirically answer the following questions: How efficiently does TroGuard trace and capture individual attributes of applications during their execution time? How accurately can TroGuard classify the applications based on the gathered labeled data logs and generate the corresponding functionality profiles? Can TroGuard distinguish between applications of different functionality classes precisely? How accurately can TroGuard estimate the functionality class that the user believes in the downloaded application is of? How does TroGuard work in details through a complete real-world case study scenario? Evaluation results will be presented regarding TroGuard's efficiency, accuracy, and discriminative power between functionality classes.

Our browser extension monitors user download activities and recommend user the most likely application type he/she is trying to download. This step is mainly done by our web page analysis tool implemented in the content script of the chrome extension. We will present the accuracy and the performance of the web page analysis in this section. The automatic application tracing and profiling are conducted both in kernel and user space. They naturally will introduce overhead for the whole system. We will discuss how the overhead looks like in this section (may be better in case study). The critical part of the TroGuard is the malware detection accuracy, also know as application type classification accuracy. We will start with evaluating different components with respect to their running environments separately. The integrated system performance evaluation will be presented in case study.

We installed TroGuard in an Ubuntu 12.10 computer system with Intel® Core™ i7 3.6 GHz Processor and 8 GB of memory was used for the experiments. We extended and used LTTng tool [57] to trace system-level activities including the semantic information such as filenames and network protocols. The Weka framework [68] was used for functionality classification of running applications.

| Web Page | CNET | Tucows | Softpedia | Download3k | Soft82 | Average |
|---|---|---|---|---|---|---|
| **Pure Text** | 0.606 | 0.209 | 0.337 | 0.334 | 0.262 | 0.378 |
| **Pure OCR** | 69.871 | 35.715 | 25.427 | 34.532 | 45.556 | 38.892 |

Table 3.3: Website analysis times (seconds)

### 3.4.1 Website Analysis Performance

Real-world usable deployment of TroGuard requires efficient and interactive website analysis so that real-time user interaction with the browser extension becomes feasible. We measured the performance of the automated website analysis engine on 100 potential download websites that include text and image files. Table 3.3 shows the performance results of applying text and image analyses for each class of websites. As shown, the website text analysis solution in TroGuard accomplishes the procedures within 0.4 seconds on average, while the image OCR analysis engines takes longer, i.e., approximately 38.9 seconds. For image heavy websites, i.e., the websites with a large number of images, such as CNET, the image analysis requires more than a minute to complete. According to the results presented in Figure 3.4, the text analysis in almost all of the tested websites produces more accurate results while running faster that facilitates the real-time interaction with the user significantly. Consequently, TroGuard makes use of text analysis unless the result are not conclusive or the website is image heavy.

TroGuard deploys the website analysis engine to simplify the users choice upon the exact functionality class, which the downloaded application is believed to be of, using the popup window that lists the ranked functionality classes. The ranked types are based on information from the webpage, such as pure text, the text in images, and interactive ads. The extension monitors the user's browsing behavior and whenever a file download link is clicked, the extension runs the web page analysis and presents

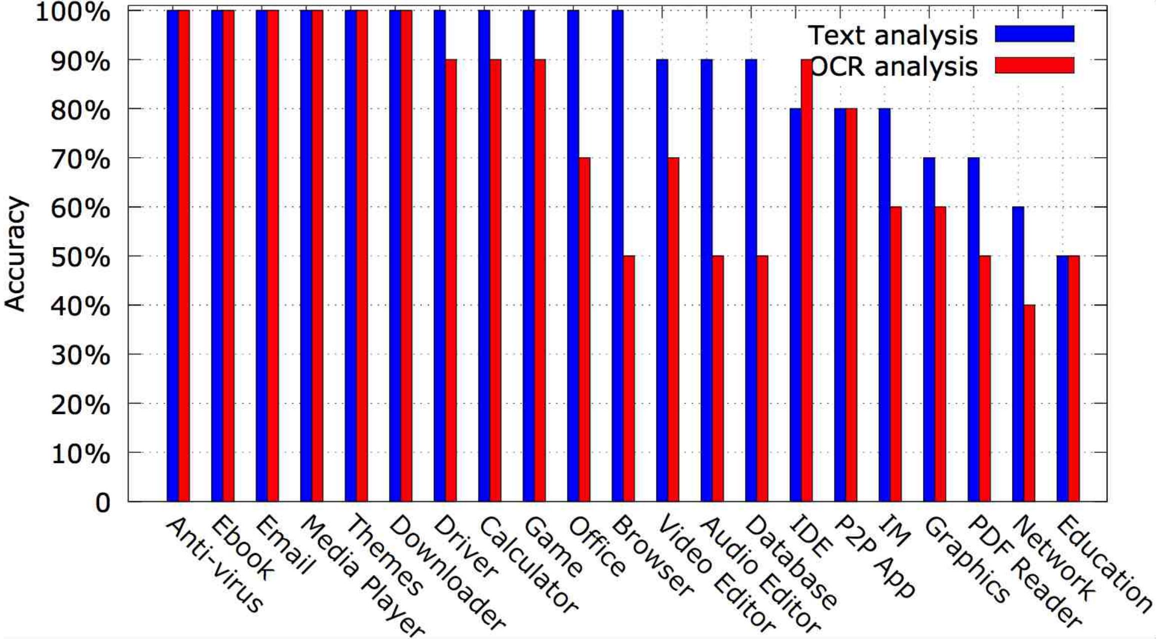


Figure 3.4: Website Analysis Accuracy

the user with a dialog box before the download starts. The user then uses the drop-down selection (see Figure 3.3) with the most likely application type ranked as its first option. The user has the opportunity to select another application type, if the default is not correct. Once the perceived functionality type is confirmed, the extension invokes (via a native NPAPI plugin component) the following tasks in our system: initiation of system-call tracing, initiation of user-space feature collection, and application classification.

We compare the accuracy of pure HTML document analysis and pure image OCR analysis by testing them in 100 popular download websites such as `Download.com`, `Softpedia.com`, `Tucows.com`, etc. We visited each download web page trying to download a random application. Figure 3.4 presents the accuracy results for the website content analysis in TroGuard. The vertical axis presents the accuracy value based on the portion of cases where a functionality class was determined correctly using the automated website analysis engine, i.e., the automatically determined class matched the functionality class that the user believed was downloading. We studied the accuracy of text-based and image-based functionality class inference separately. For image-based analysis, TroGuard uses the open source Tesseract OCR engine [61]. In particular, the engine took screen shots of the download website before performing an in-depth OCR analysis to search for the target key terms, e.g., the functionality class names and related keywords. It is noteworthy that the OCR analysis accuracy is relatively lower compared to the pure text-based analysis. As shown in the figure, TroGuard's pure text-based analysis engine outperforms the image-based website analyzer in almost all of the cases except the IDE functionality class.

### 3.4.2 Application Classification Accuracy

To evaluate the application classification in TroGuard, we installed 100 applications[1] chosen from ten different functionality classes (10 application per functionality class) and collected data corresponding to individual attributes. During the off-line training stage, each application ran for 60 seconds while a user exercised the application's main functionalities. It is noteworthy that, after several trials, we found the 60-second interval sufficient to capture the main functionalities of each application according to its functional category. Each application's data log was divided into six time periods (10 seconds each), to be used later for testing phase. Consequently, our collected database composed of 600 labeled data points.

We evaluated the performance of application type learning and classification component in TroGuard using 10-fold cross validation [69]. We considered two statis-

[1] The applications were chosen from Ubuntu Software Center and www.Softpedia.com.

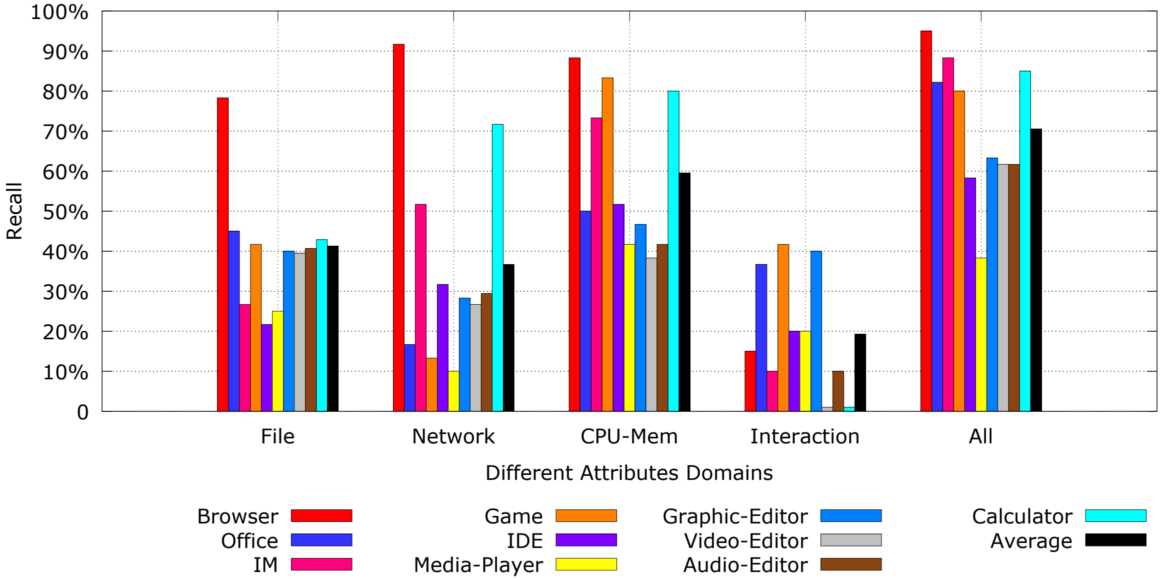


Figure 3.5: Classification recall for detectors using different attribute sets on different functionality classes.

tical measures, namely 1) recall, the fraction of relevant instances that are retrieved during the classification; and 2) precision, the fraction of retrieved instances (i.e., marked as a particular functionality class by TroGuard) that are correct (i.e., actually of the marked class).

Figure 3.5 shows the recall values for each functionality class as a result of classification using different subsets of attributes. The employed attributes fall into four categories, file system-related (File), network-related (Network), system resource-related (CPU–Mem), and user interactivity-related (Interaction). We also present results of the classification using all of the attributes. The results for each subset of attributes are a clustered set of bars, each representing the recall value for an individual functionality class. As the results show, the browser functionality class gives the highest recall percentage except when the classification is performed using the Interaction attributes only. This is because various web pages require different kind of user interactions. As a case in point, the way users interact with an online game website,

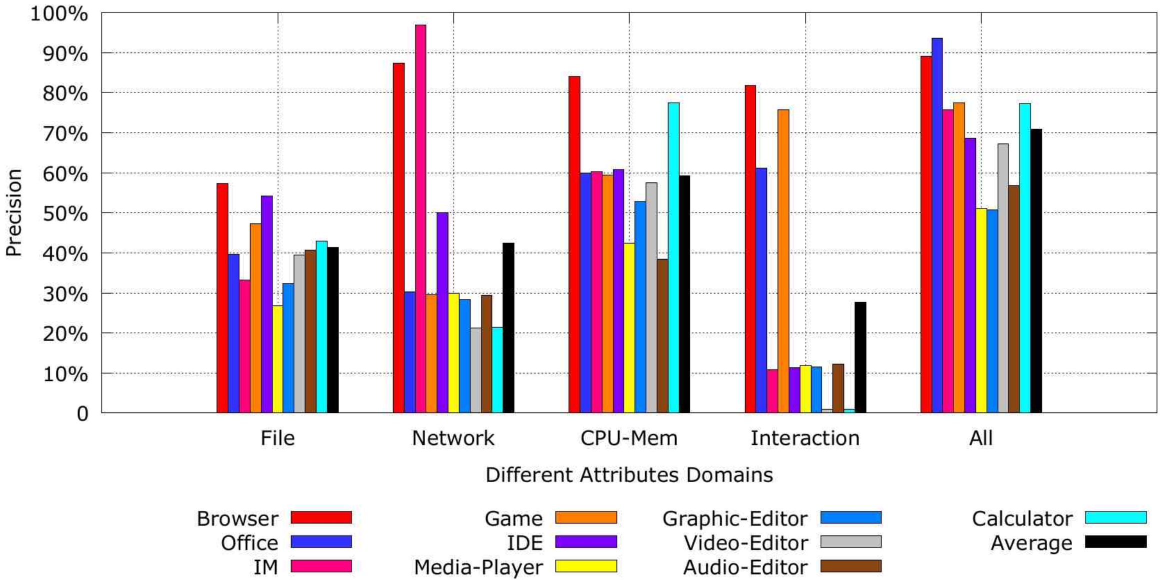


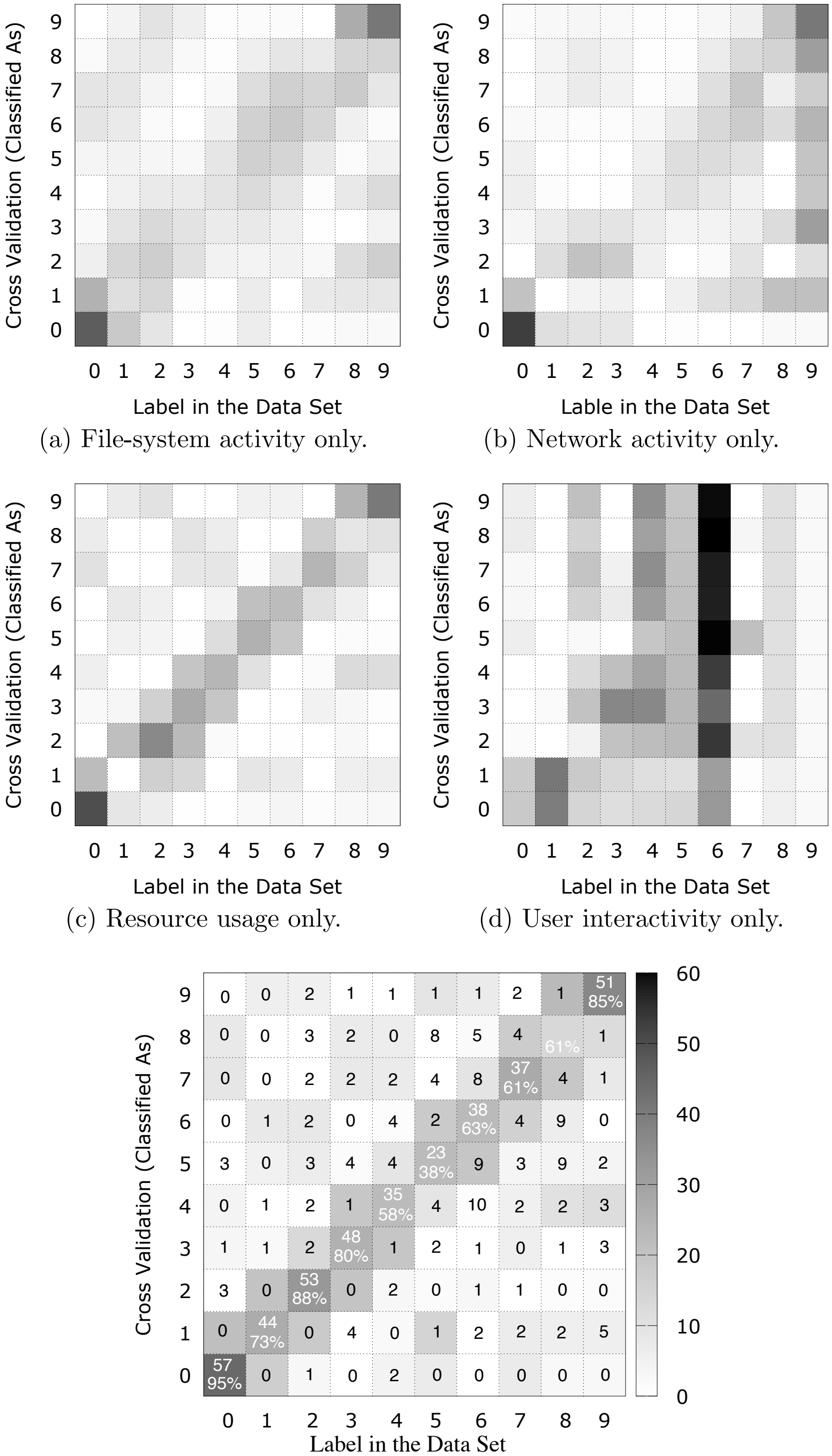
Figure 3.6: Classification precision for detectors using different attribute sets on different functionality classes.

i.e., intense keyboard and mouse interaction, is extremely different from that of a text or video news channel, i.e., minimal interaction while the video is being played. Therefore, many browser samples have been wrongly marked as other functionality classes such as graphic editor that require similar mouse activity. Another interesting functionality class is the game class whose most distinguishing characteristic are its CPU–Mem attributes because games usually require high CPU and memory resources in comparison to other functionality classes. The network attributes are the weakest for distinguishing game samples as they do not make regular connections to network very often.

Figure 3.6 also shows the precision results for the application type learning and classification component of TroGuard. As can be seen, using all of the attributes results in application marking with a higher accuracy. Comparing the recall and precision results using Interaction attributes for the browser class, the classifier has been selective in marking application samples as the browser functionality class resulting in high precision. Additionally, many of the graphic editor, media player and IDE samples are marked falsely as browsers resulting in a poor precision rates. Based on both recall and precision results, the user interactivity attribute contributes the least to the overall accuracy. In addition to the use of different GUI schemes in applications, this may be attributed to several applications executing in the background while they are not the active window on screen, i.e., zero user interactivity.

Figure 3.7 shows the accuracy results from a different perspective, providing more details on how applications were classified correctly or confused by mistake by the TroGuard framework. In particular, we present one $10x10$ gray-scale confusion matrix for each set of attributes, e.g., File. In the confusion matrix, a darker shade of gray in a cell indicates that the detector had higher confidence in classifying a program from the given row as the part of the functionality class from the given column. A perfect detector produces a completely black diagonal confusion matrix. To further clarify, each $(i, j)$ element in a confusion matrix represents how many of application data points of the $i$-th functionality class was marked (classified) as the $j$-th functionality class[2]. Therefore, the diagonal elements denote the samples that were classified correctly. Clearly, out of the four sets of attributes, CPU–Mem attributes are the most discriminative, while Network attributes do relatively well except for the data points labeled as the calculator functionality class. Regarding the least contributing attribute set (Interaction), as the Media Player applications do not require high user interaction, most of their corresponding data points (column 6) have been classified wrongly. On the other hand, we can see accurate classification results for IDE applications with high user interaction.

[2] The functionality class numbering is as follows: Browser (0), Office (1), Game (2), IDE (3), IM (4), Graphic Editor (5), Media Player (6), Video Editor (7), Audio Editor (8), and Calculator (9).



(a) File-system activity only.

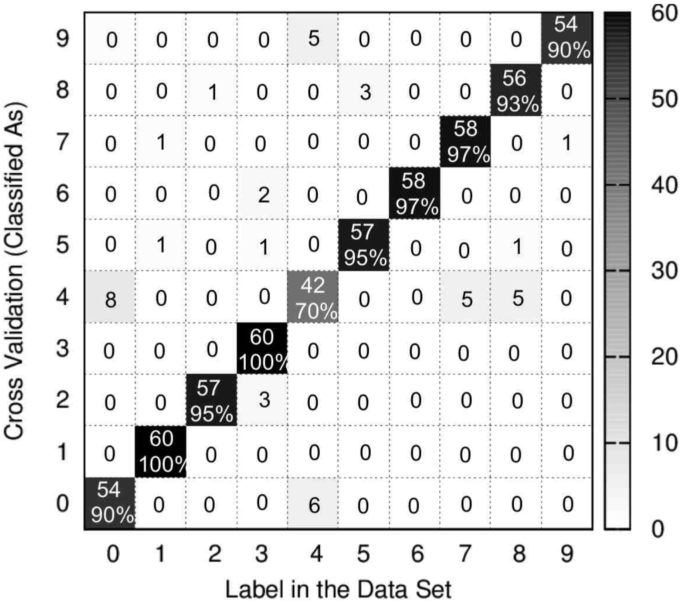
(b) Network activity only.

(c) Resource usage only.

(d) User interactivity only.

(e) All activity attributes.

Figure 3.7: Confusion matrices for a variety of detectors. Figures (a)–(d) describe class-specific detectors, each using a particular class of activity attributes. Figure (e) describes a detector using all attributes.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 54<br>90% |
| 8 | 0 | 0 | 1 | 0 | 0 | 3 | 0 | 0 | 56<br>93% | 0 |
| 7 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 58<br>97% | 0 | 1 |
| 6 | 0 | 0 | 0 | 2 | 0 | 0 | 58<br>97% | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 0 | 57<br>95% | 0 | 0 | 1 | 0 |
| 4 | 8 | 0 | 0 | 0 | 42<br>70% | 0 | 0 | 5 | 5 | 0 |
| 3 | 0 | 0 | 0 | 60<br>100% | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 57<br>95% | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 60<br>100% | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 54<br>90% | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 |

Figure 3.8: Classification Using Intermediate Features

Finally, we measured how much the usage of intermediate-level features improved the classification accuracy (Figure 3.8). For intermediate-level feature-based functionality classification, TroGuard uses Debtags [58] that is a large set of Debian packages, approximately $39K$ packages, each labeled manually with many *facet* features by developers to facilitate application package search by high level application functionality tags. We grouped the packages into 10 functionality classes automatically using logical rules on combination of facet features, and then verified the automatically categorized packages one by one manually for correctness. As a case in point, we used the following rule for the browser category `browser:|use::browse|format::html|protocol::http`, i.e., the package is a browser type if its facet feature set include any of the three listed features. We later used the abovementioned functionality class labels for supervised learning using the intermediate-level facet features of each Debtags data point (application). The computed learning rules were later used for trojan detection and classification of legitimate Debtags and non-Debtags applications. The final average accuracy was 96% compared to 70% (see Figure 3.7) that empirically proves the benefit of intermediate-level features as an approach to fill the semantic gap between the high-level functionality classes and low-level system traces.

### 3.4.3 Application Classification Performance

During the application type learning phase, TroGuard saved the collected data for the 100 applications in human readable plain text files (totaling 40.5 GB). TroGuard parses each file, extracts and stores the suitable attributes for the application classification step later.

TroGuard built 5 models using the meta classifier Ensembles of Nested Dichotomies (END) in the Weka machine learning suite [68], based on the C4.5 decision tree classification algorithm [68]. The algorithm can handle both continuous (numeric) and discrete (nominal) attributes. TroGuard builds the decision tree with 2 as the minimum number of instances per leaf. Additionally, TroGuard uses 1 fold for reduced-error pruning, and 2 extra folds for growing the tree. Table 3.4 shows the performance results for the application classification engine. As shown the total 81 attributes were distributed among the four attribute sets. The second row shows how long it takes for TroGuard to complete the supervised learning and training phases using all of the application data logs. The first four columns show the time requirement for classification using a subset of attributes, and the last column reports the overall time required for creating trained models using all of the attributes. As shown, the training phase using the all the attributes takes less than 1 second to complete that is an acceptable duration for real-world deployment of the TroGuard framework.

Table 3.4: Training times (seconds)

| | **File** | **Network** | **CPU-Mem** | **Inter.** | **All** |
|---|---|---|---|---|---|
| **#Attr.** | 44 | 20 | 8 | 9 | 81 |
| **Time** | 0.49 | 0.14 | 0.19 | 0.4 | 0.82 |

In the evaluation of the accuracy of the model, we divided our feature vector(data attributes) into four categories, file system related attributes, network related attributes, resource usage related attributes, and user interactivity attributes. We took the attribute as a variable and to use cross-validation to verify how those different domain information effect the accuracy of the model. File system related feature has been broadly explored in the past [70], [55], etc.. Application profile that heavily rely on file system related feature is effective in detect malicious program functionalities in some cases, but those system is easy to be circumvented and very often they fail to detect the new malicious application. As we will present, combining with other features independent of file system is effective to distinguish malware and benign applications.

### 3.4.4 Sandboxing Performance

We measured how much overhead TroGuard's generated SE-Linux policies put on the system once the user downloads the application. Figure 3.9 shows the results. On average TroGuard's sandbox puts 5.4% overhead on the system throughput. It is noteworthy that the overhead is temporal and will go away once TroGuard confirms that the application is benign and could run outside of the SE-Linux sandbox.

### 3.4.5 System Performance Overhead

We studied the performance overhead of data collection tool LTTng kernel tracer. The LTTng kernel tracer is a highly efficient tracing tool that typically used by

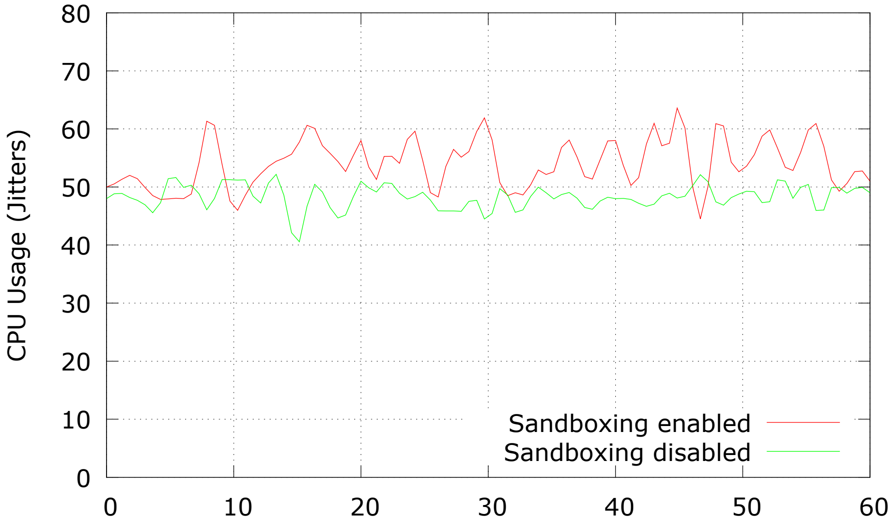
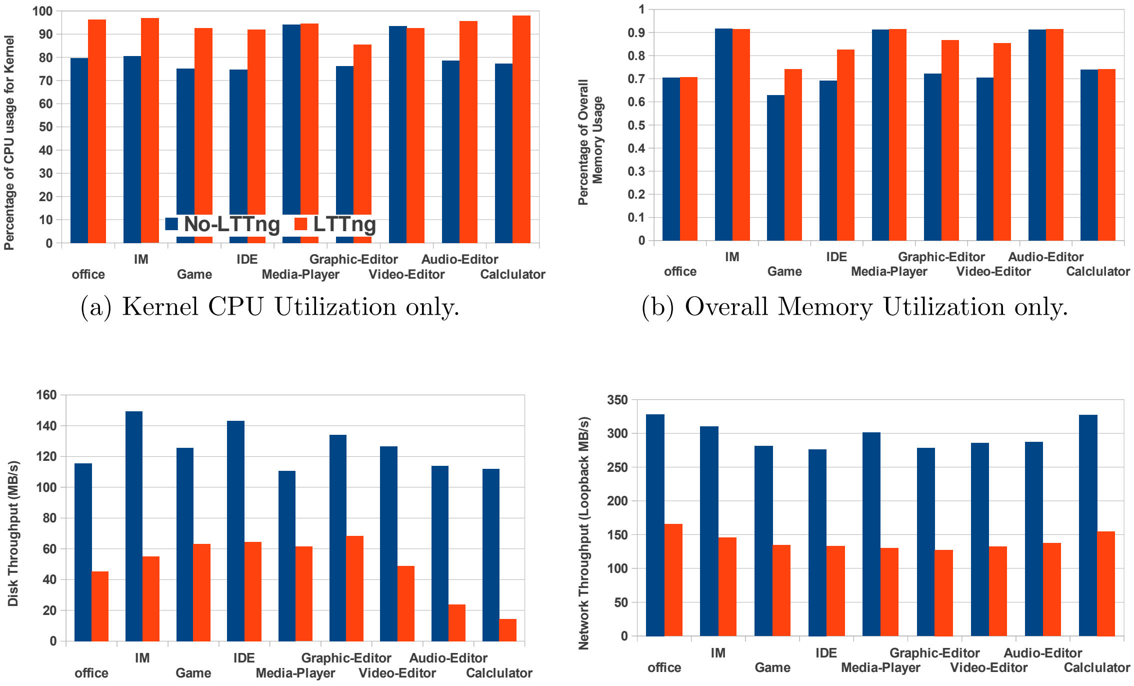


Figure 3.9: Sandboxing CPU Usage Overhead

system developer to track down performance issues and debug problems in multiprocess system. The reported impact of LTTng on kernel operations in terms of percentage of CPU cycles against vanilla kernel is less then 5% in [71].

We measured the impact of TroGuard syscall interception engine on various aspects of the system performance, namely kernel CPU utilization, memory utilization, disk throughput, and network throughput. In addition, we benchmarked the network throughput over loopback. Figure 3.10 shows the results. In our experiments, the maximum CPU overhead TroGuard's kernel tracer introduced is 20% for the calculator. The maximum memory overhead is 15% for running video editor. The measured performance overhead for disk throughput ranges from 45% to 87%. The fairly high disk operation overhead of TroGuard is due to its logging activities to the system's disk in parallel. The network performance degradation ranges from 55% to 60%. It is noteworthy that these overheads are not permanent and go away once TroGuard comes to a conclusion on whether the running application is a trojan.



(a) Kernel CPU Utilization only.

(b) Overall Memory Utilization only.

(c) Disk Throughput (Loopback) only.

(d) Network Throughput only.

Figure 3.10: TroGuard Overhead on System Resources

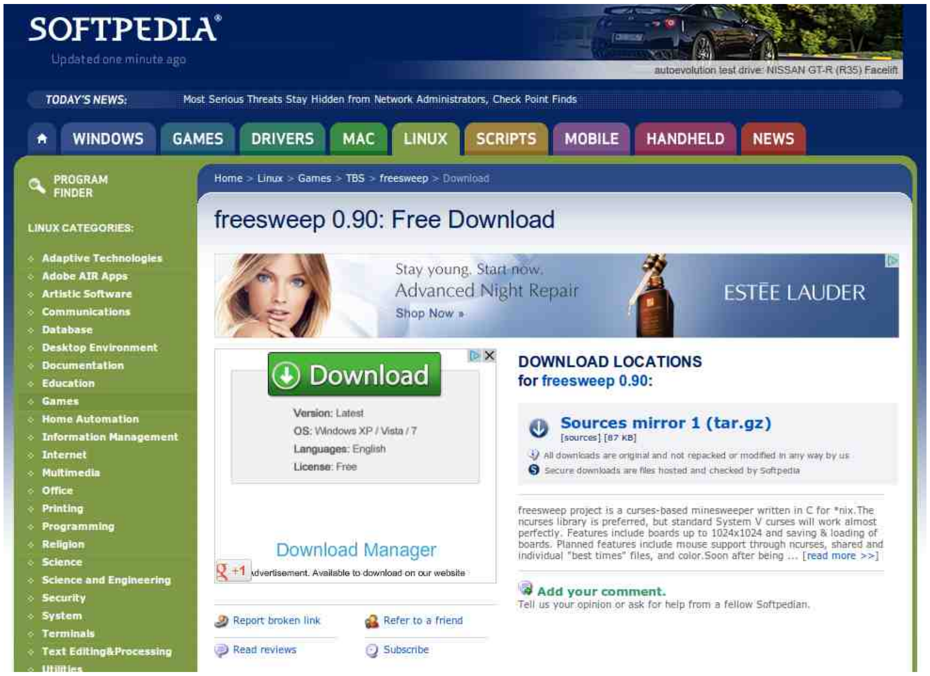
### 3.4.6 Behavioral Mimicry Attacks

One of the main challenges that behavioral-based detection solutions face generally is their vulnerability against behavioral mimicry attacks. We evaluated how TroGuard performs in terminating such attacks that try to pretend delivering a legitimate functionality while conducting a malicious activity in the background. First, we describe a concrete case study to show step by step how TroGuard detects a sample trojan `Freesweep` game application that is downloaded from a legitimate-looking malicious website. Next, we will present our results on 50 other mimicry attack samples.

We obtained the `Freesweep` game application trojan payload using the Metasploit suite [72] as a Ubuntu deb game package. The trojan package included the original game, the exploit payload, and a post-installation script that executed the exploitation script. Upon execution, the malicious payload opened a socket connection secretly and created a reverse shell to connect back to an adversarial remote site.

We masqueraded a `Softpedia` web page as shown in Figure 3.11 to hold the malicious trojan game package. Consequently, the victim user was convinced to download, install and run the game on her desktop computer. Once the victim clicked to download the package, TroGuard's extension component noticed the download action request in the browser. TroGuard analyzed the download website and correctly estimated the functionality class, i.e., game. The determined functionality class was shown to the user through a browser popup (similar to Figure 3.3) with the default option set to game. After the user's confirmation, TroGuard switched to the monitoring mode to trace and capture the application's execution footprint. Analysis of the logged data sets using the 100-application trained model resulted in classification of most of the downloaded application's data points as calculator. Figure 3.12 shows the calculated probability values for each functionality class, i.e., the probability that the downloaded application is of a particular functionality class. The mismatch between the confirmed functionality class and the derived class caused TroGuard to raise a warning notifying the user of the mismatch and a potential malicious download.

We evaluated TroGuard on 50 mimicry trojan samples created by reverse engineering and malicious code patching of the benign pdf readers. Figure 3.13 shows the classification results of how accurately the mimicry trojans are classified compared to their corresponding benign applications. The ratings are averaged over all trojan samples for each classification parameter that show lower classification accuracy when TroGuard classifies a trojan. For instance, TroGuard classified 90% of benign



freesweep 0.90: Free Download

Version: Latest
OS: Windows XP / Vista / 7
Languages: English
License: Free

Advertisement. Available to download on our website

DOWNLOAD LOCATIONS
for freesweep 0.90:

Sources mirror 1 (tar.gz)
[sources] [87 KB]

All downloads are original and not repacked or modified in any way by us
Secure downloads are files hosted and checked by Softpedia

freesweep project is a curses-based minesweeper written in C for *nix.The ncurses library is preferred, but standard System V curses will work almost perfectly. Features include boards up to 1024x1024 and saving & loading of boards. Planned features include mouse support through ncurses, shared and individual "best times" files, and color.Soon after being ... [read more >>]

Add your comment.
Tell us your opinion or ask for help from a fellow Softpedian.

Figure 3.11: Game Trojan's Download Webpage

pdf-reader applications correctly, in contrary, only 10% of mimicry trojans that pretended to be a pdf-reader application was classified as a pdf-reader by TROGUARD. All others were classified as Because of such low classification accuracies (below the predefined acceptance threshold 0.8), TROGUARD was able to detect and terminate all 50 trojan samples.

### 3.4.7 Security Analysis and Discussion

We discuss various aspects and the potential deployment limitations of the TroGuard framework.

How representative are the functionality classes in TroGuard? Coming up with a comprehensive, meaningful and universal set of functionality classes is challenging, and at the same time, critical to the TroGuard's correct operation. To this end, we have identified and used a common set of functionality classes from three popular software-download web sites and the Debian package tags that host a large compre-

| *Functionality-Class Mismatch:* | |
|---|---|
| Expected: | Game |
| Observed: | 10% Game, 50% Calculator |

Figure 3.12: TroGuard Alert

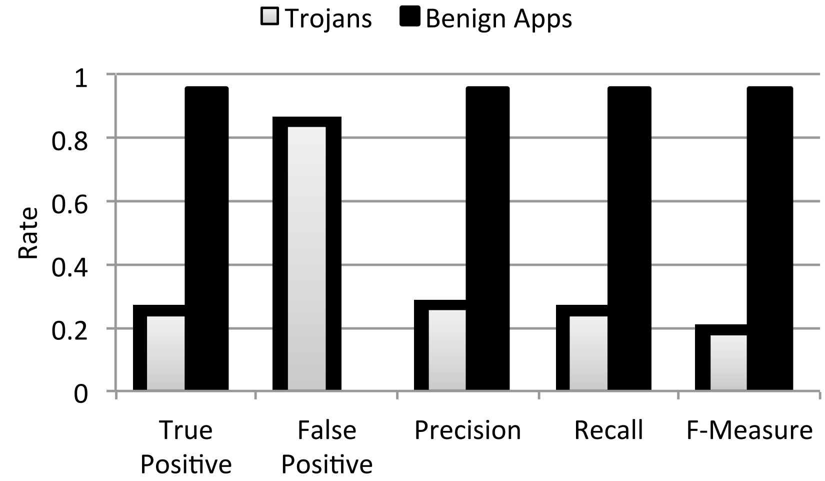


Figure 3.13: TroGuard Against Mimicry Exploits

hensive set of software applications and are viewed by millions of users every day (Section 3.2.1).

How usable can TroGuard be for non-technical users? To provide the users with a usable security solution, one of our main objectives in designing every component in TroGuard has been to separate the low-level system details that are used for various analyses from the high-level information that is communicated to the user. As a case in point, while the kernel-level modules are tracing different system-level activities of the suspicious application, the user's interaction with TroGuard is through the browser extension using terms that the (potentially non-technical) user usually sees on popular download websites and is familiar with. In the simplest form, the user only has to confirm that the application class diplayed by the browser extension matches the class he expected based on what he saw and read on the download page. We believe that such design points has made TroGuard usable by a vast range of users. Of course, this belief has to be validated through an actual usability study, and we intend to pursue such a study in our future work, pending IRB approval.

Is TroGuard useful even if it is not absolutely accurate? Semi-automated detection of social engineering attacks using mostly system-level information has remained to be a very challenging problem that is also indicated by their recent increasing popularity among attackers. TroGuard introduces a new solution against web-based socially engineered trojan attacks through categorization of applications' functionality classes rather individual applications (developed by possibly unknown third-parties). Additionally, as our implementation is not yet optimized, both the website analysis and application classification accuracies can be improved through parameter tuning as well as usage of more complete techniques such as static executable analysis techniques to estimate the application's functionality class.

How can attackers evade the TroGuard's detection analyses? Evasion of the TroGuard's detection solution requires the trojan application to accomplish its malicious tasks, such as sensitive system file modifications, through complete imitation of a benign functionality class. This is a feasible attack against not only TroGuard but also most of the anomaly-based malware detection solutions in general, and we note that detecting mimicry attacks is a hard problem and an ongoing research topic.

What if TroGuard notifies the user about a downloaded trojan too late? To minimize the possibly of late user notification, TroGuard performs real-time analysis of the execution and warns the user as soon as its confidence regarding the ongoing malicious activity gets high enough. Furthermore, to eliminate the possibility of late notification, sandboxing could be deployed once the application is installed according to its functionality class. For instance, a password file access attempt by a game trojan application would be blocked by the deployed sandbox. We consider this extension outside the scope of this paper and consider it as a future work.

# CHAPTER 4

# TroGuard as a Cloud-Based Malware Detection Service

## 4.1 System Overview

We present the design and architecture of the proposed system in this section. Our goal is to create a vertical malware analysis and detection framework that is efficient, transparent to endpoint users, and with truthful detection capability. To this end, the system includes two major components as shown in Figure 4.1: a user agent and a cloud analysis platform. The user agent in our design is integrated into a web browser as an add-on application to provide on-demand malware analysis interface to the cloud. However, this is not the only use case possible. The user agent could also be implemented in security gateways as a traffic processing plug-in to detect malicious files in a network. The following discussions of user agent focus on endpoint user, while the design principle of the two are very much alike. The cloud analysis platform consists of database systems, dynamically provisioned virtual sandboxes, and computation units that enable malware sample storage, context aware behavioral analysis, and efficient behavior feature modeling and classification.

The highlight of the system is that it introduces an advanced on-line learning algorithm to learn the best analysis time frame achieve efficient analysis and accu-

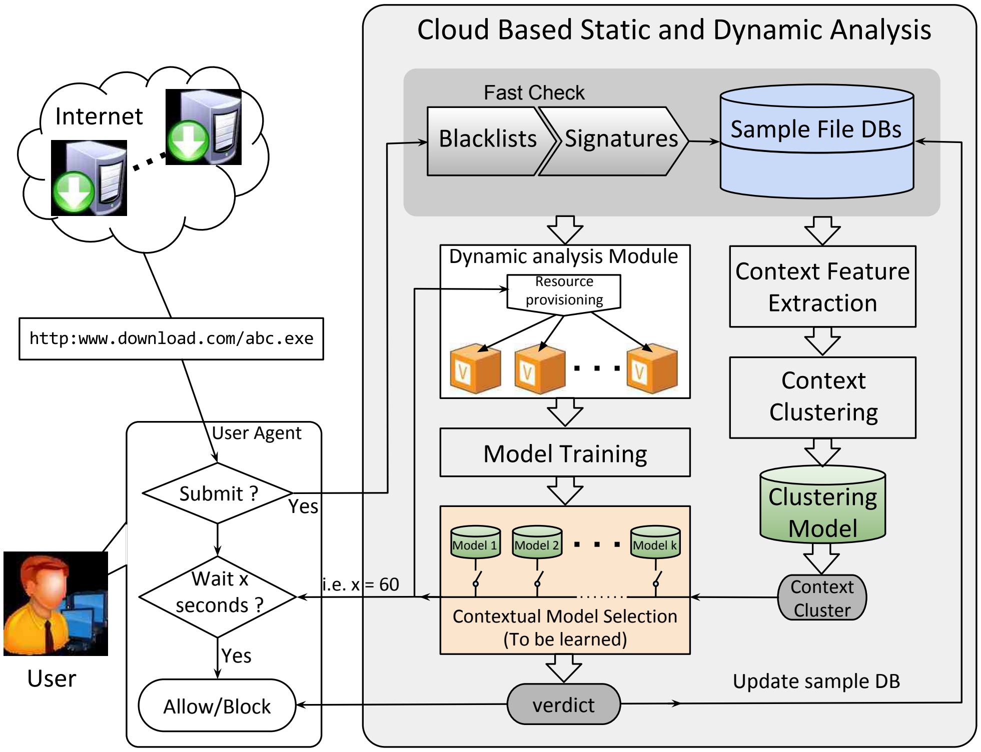


Figure 4.1: System Architecture

rate detection of heterogeneous malware samples. In contrast, all existing behavior analysis systems select the time length of dynamic analysis using some heuristics and apply uniformly to all samples. Our improvement also enhance the system usability by providing users with optimal waiting time (shortest waiting to get the most accurate result) and a projection of the low bound for detection accuracy. All the information will be available before the cloud start the dynamic analysis, making it much user-friendly because it offers users the flexibility to configure the length of sample execution (may receive suboptimal analysis result). In the following sections, we will discuss each component in details.

### 4.1.1 Client Agent

In a system protected by our service, when a user clicks a URI to download an executable file within a browser, the installed user agent is activated and pop up a window to offer him/her the cloud based malware detection service. If the user click accept to scan the file, the client agent will connect to the remote file downloading server and start the file download. Instead of writing the file data to local disk, the client agent will initiate a separate connection to the cloud analysis platform and forward the downloading data to it. This design leave no chance for the suspicious file to infect the host computer because the client agent is run in a sandbox environment. The packets received at the cloud are reassembled into executable files when all the packets in the downloading session arrived.

In each detection task, client agent keep interacting with cloud system until received the final verdict. The first message from cloud informs user of the required dynamic analysis time, computed by the contextual bandits algorithm according to the samples context information. Users are free to decline or accept the waiting. In the former case he/she receives no malware protection, while in the later case the user have to spend extra time to download, allowing an extended period of dynamic analysis for detection results. To reduce the length of waiting is very important for improving user's quality of experience (QoE), therefore our learning strategy at cloud is designed to optimizing the length of dynamic analysis (major contribution to the waiting) with the condition of accurate detection and efficient resource usage. As noted above, client agent could also be implemented as a plug-in of security gateway systems configured by the IT administrator.

### 4.1.2 Malware Analysis as a Cloud Service

The malware analysis and detection engine is designed to be deployed in a cloud environment to reach scalability and performance requirements and to meet the user's QoE. The platform could be deployed in a private cloud to protect an organizational network or in a public cloud to provide malware detection services for all the subscribed users. The cloud based detection engine includes fast file checker, context feature extraction and clustering modules, dynamic behavior analysis module, and contextual multi-armed bandit learning and classification modules. Figure 4.1 shown the modules and the data flow of the system.

**Fast Check Using Blacklists and Signatures.** The Fast Check is the first module processes the received executable file. It use traditional pattern machining method to fast check the known malware to reduce unnecessary delay. We use Safe Browsing URI lists as the blacklist to check against. It reduces detection latency for some well known malicious web resources, If a user try to submit a executable download from a URI that is in the Safe Browsing database, the user will experience minimum delay in receiving the detection result. If this is the case, the sample and its label will be stored in the sample file database for future model update. Otherwise, the downloaded file will be passed to the signature scanner.

Using signature scanning alone cannot defeat new malware, but like using blacklist it is effective for filtering samples known to the signature database and reduce system load. The difference between signature scanning and blacklist checking is that signature scanning matches file signatures against file content while blacklist checking the requested URI against entries in the list database. We use a collection of antivirus products (via Virus Total APIs) to scan the submitted files to avoid repeated analysis of same file. It combines results aggregated from 54 anti-virus engines and we use majority vote to drawn the conclusion, i.e., we treat the files that marked by half of anti-virus engines as malware and return the result to user agent. Fast Check module is also a labeling tool that generates labeled dataset for training classifiers for malware detection.

**Context Feature Extraction.** In our real-life malware samples, we observed some distinctive file characteristics that separate the malicious samples apart from the benign ones. For example, malicious executables are usually packed or obfuscated while benign programs are not; the average size of malicious executables is far smaller than the average size of benign programs, etc.. These file characters are not sufficient to be used as features in training reliable detection models; However, they can be used to facilitate the behavioral feature analysis and classifier selection. File characteristics are viewed as context information in our work and is the key of the contextual multi-armed bandits modeling (Section 4.2). By following a designated learning policy, the system learns the best mapping from context information to the classifiers that trained using behavioral features from different analysis length. Our assumption is that only those samples with similar context features should be included in training an accurate behavioral feature based classifier. In other words, classifiers need to be trained separately according to their context feature. After all, it is not reasonable to compare the behavior features of packed binaries and those unpacked ones.

**Context Clustering.** Initially, cloud will make use of context features of the training samples to build an clustering model that divide the context features space into subspaces. The purpose of this context space partitioning is to allow the system to learn the performance of different classifiers for a subspace of contexts rather than each individual context, thereby improving the learning speed significantly.

Context clustering model is trained using the context features from the submitted samples that is known to the Safe Browsing and anti-virus signature scanning components. For new samples submitted for detection, context cluster labels are revealed first and applied to the Contextual Model Selection module to select the best classifier from array of classifiers. This made possible by the learned mapping relation between context context information and the best classifier given the context. There are two advantages of introducing the context clustering module. First, the easy obtained context information determines how long it need to execute the sample in order to use the selected classifier, as the classifier is trained with behavior features from same length of execution. Secondly, the determined optimal execution length feedback to the user and make our system much user-friendly than conventional systems. Specifically, users of our system are explicitly informed of how much delay he/she would experience and given the option to alter it, while users of all other systems suffer from longer waiting for all the detection tasks.

**Dynamic Behavioral Analysis.** The Dynamic Analysis Module is equipped with virtualized sandbox to conduct on-demand dynamic behavior feature collection. We modified Cuckoo Sandbox [73] by adding an interface to the Context Clustering Module so that the determined execution length for the sample can be passed to the Resource Provisioning submodule to time the sample execution. The Resource Provisioning submodule dynamically allocate Virtual Machine (VM) instances based on individual requests. Every submitted sample does not hit the fast checker database will be run in an instrumented VM dedicated for the particular sample. The virtual resource of each analysis instance such as CPU cores, base memory, hard drive, video memory, etc., is a variable because different sample requires different resource requirements. However, the available physical machines that host the instances is limited by the overall infrastructure. For a cloud system with thousands of users, the VM instance provisioning process need to be carefully planned to avoid wasting the computing resources — i.e., allocating too much disk space for one analysis instance that will not be fully used during the sample execution. One strategy we have already introduced is to combine context feature clustering in deciding the optimal tracing length. This has to do with the CPU time. In later section we will introduce the definition of QoE which will take into account the memory consumption for executing a sample in order to balance the trade off between the achieved accuracy and resource cost it takes. The objective is to improve the QoE for users.

Once the executable start executing, a script emulating a human user will start clicking the software's GUI to cover more functionality of the sample software. We have enhanced Cuckoo's out-of-box human emulation script by adding capability to search and click UI buttons in Chinese and Russian, because many of the studied samples have user interface in the two languages other than Cuckoo Sandbox's default English.

**Detection Model Training.** Unlike most existing dynamic behavioral feature based malware detection systems that employ a single detection model with fixed length of behavior monitoring, our cloud based detection system maintains multiple models trained with behavioral features collected from different length of executions. Specifically, each training sample will be execute multiple times with different execution length denoted by a discrete set $\{\tau_1, \tau_2, \ldots, \tau_K\}$. A finite set of supervised learning models $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ have been trained and readily to be deployed through *Contextual Model Selection* module. The training samples could be a mix of legitimate executable files and some historical malware samples that have been manually analyzed by anti-virus organizations or other malware researchers. The implementation of *Model Training* module lies in the field of supervised machine learning. Existing algorithms could be applied to search in a function space $f_k : \mathcal{X}_k \to \mathcal{Y}$ for a detector with least cross validation error. Here $\mathcal{X}_k$ is the feature space obtained from $T_k$ period of dynamic analysis of submitted samples and $\mathcal{Y}$ is the label space. The function space searching is well studied in supervised machine learning and malware classification literatures [19, 25, 32, 74, 75]. So that we will not discuss the details about model selection and hyper-parameter searching in this work, instead we design cloud interfaces that allow any independent implementation of the *Model Training* subsystem to be integrated in to the cloud platform.

**Multi-Armed Bandit Learning.** We introduce the contextual multi-armed bandit learning framework to learn the best classifier (require shorter period of dynamic analysis) based on sample's context information. Learning the best classifier among many is necessary because unlike applications such as speech and text recognition where audio and image features remain relatively constant over samples, malware behaviors evolve and some times adversaries attempt to fool the detectors by delaying the malicious activities. So that it is necessary to maintain multiple models with different length of behavioral profiles and allow the system to choose the best classifier in order to achieve high accurate of detection by capturing more behavioral activities. We will presented the complete model of the framework in the next section and discuss the details of the algorithm we proposed in order to achieve highest expected QoE.

## 4.2 System Model

Behavioral feature based malware classification usually modeled as a supervised learning problem in the past. Under the framework, a malware classifier $f$ that trained with labeled history feature vectors will be applied to the vectorized features to compute the likelihood of the sample being malicious. We noted that malware behavior features are highly depends on the length of monitoring $\tau$, the performance of the classifier in turn depends on $\tau$. Generally, larger $\tau$ leads to accurate classifiers, while smaller $\tau$ gives classifiers that perform worse. To improve the performance of the malware classifier, increase the length of behavior monitoring $\tau$ arbitrarily will drain the computation resource of the cloud system. In practical system, we need to carefully balance the trade-off between achieving excellent accuracy and the incurred cost, both of which connected to $\tau$. Our proposed system maintains multiple classifiers $f_{\tau_1}, \ldots, f_{\tau_k}$ trained with behavior features from different monitoring period $\tau_1, \ldots, \tau_k$. For each individual detection task, the system learns in real time which classifier is the best one to choose. We model such a learning process as a contextual multi-armed bandit problem.

In this section, we focus on the modeling of the *Contextual Model Selection* module in Figure 4.1. The problem of time-variant behavioral feature based malware detection system can be naturally modeled as a multi-armed bandit problem with malware context information.

### 4.2.1 A On-line Classifier Selection Problem Formulation

The original multi-armed bandit setting includes a finite set of $K$ actions $A = \{a_1, \ldots, a_K\}$. In each round $t = 1, \ldots, T$, one particular action $a_k$ is taken and the corresponding reward $r_t(k)$ for the action will be returned. The reward $r_t(k)$ is chosen from a stationary probability distribution that depends on the action $k$. The goal is to design a policy that maximize the total rewards through repeated action selection.

If there are contextual $\mathbf{z}_t$ available at time $t$ to assist the action selection, the problem becomes a contextual multi-armed bandit problem.

The problem of malware classifier selection can be naturally modeled using the contextual multi-armed bandit framework outlined above. The *Contextual Model Selection* module from Figure 4.1 maintains a finite set of malware classifiers $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ indexed by $\mathcal{K} = \{1, 2, \ldots, k\}$, for which each classifier $f_k \in \mathcal{F}$ is trained off-line with behavioral features from a specific execution time $\tau_k$ and associated with an unknown and fixed accuracy distribution $\mathcal{D}$ over $[0, 1]$. In the system introduced in Figure 4.1, consider the most recent $N$ files that have been submitted to the cloud along a discrete time horizon $t$ by either personal users $a^t$ via web browser or security gateway systems. As part of the requests handling process, the *Contextual Model Selection* module will select and apply one of the $k$ classifiers to the given sample's behavioral feature vector $\mathbf{x}^t$ to output a classification result $y^t = f_k(\mathbf{x}^t) \in \mathcal{Y} = \{0, 1\}$. This corresponds to choose an arm to play in original bandit problem. In our model, the reward received for the module by selecting $f_k$ is an indicator function $r_t = \mathbb{1}(y^t = \hat{y}^t)$, in which $\hat{y}^t \in \mathcal{Y}$ is the true label of the sample. The "1" in the binary label set $\mathcal{Y}$ represents malware and "0" for legitimate software. In practice, the detection result $y^t$ could also be a probability prediction (e.g. $y^t \in \mathcal{Y}' = [0, 1]$, values from the range represent lowest to highest possibility of being a malware) . It is worth noting that the feature vector $\mathbf{x}^t$ in round $t$ and the training features of classifier $f_k$ come from behavioral features collected during dynamic execution for time length of $\tau_k$.

Each of the $k$ classifiers has an expected or mean reward given that it is selected in multiple rounds, we call it the *accuracy* of the classifier. We denote the classifier selected on time step $t$ as $F_t$, thus the *accuracy* of an arbitrary classifier $f_k$ denoted $q(f_k)$, is the expected reward given that $f_k$ is selected:

$$q(f_k) = \mathbb{E}[r_t|F_t = f_k]. \tag{4.1}$$

The *accuracy* is a simple and intuitive metric to evaluate classification system. As a matter of fact, majority of dynamic feature based malware classification systems presented evaluation result in the similar form of measurement: precision, recall, F1 score etc., while ignoring the cost incurred in conducting dynamic behavioral features analysis. We observed that in a user interactive system under limited computation resource budget, to achieve the ultimate behavioral based classification accuracy through comprehensive dynamic analysis is impractical. Thus we designed a new metric Quality of Experience (QoE) with the mind of balancing the trade-off between high analysis accuracy and the cost of analysis.

**Definition 1 (Quality of Experience).** The Quality of Experience received by user $a^t$ at time $t + \tau_t$ by selecting the $k$th classifier from $\mathcal{F}$ at time $t$ is the weighted sum of the classifier's accuracy and the incurred cost because of $\tau_t$ length of dynamic analysis

$$Q(f_k) = q(f_k) - \beta c(\tau_k) \tag{4.2}$$

Where $\beta \in [0, 1]$ is a trade-off parameter that depends on the application requirements. ■

We now have the QoE as a measurement of how the cloud based detection system performs. Briefly, we want to maximize the expectation of QoE by determine how long to execute each sample in order to apply one of the maintained classifiers based on their evaluative feedbacks, i.e. the history QoEs. As we don't have the true value of $Q(f_k)$, we have to estimate it for each $k$ in order to find the maximum. One possible method to do this is to start with large value of $k$ to explore as many behavioral features as possible until a superior execution length (smallest $k$ that result in highest empirical mean of QoE) is observed among $\mathcal{K}$, and switch to the particular choice of optimal value $k^*$ to exploit the benefits of fast and accurate detection. However, this greedy method subject to sub-optimal result because in all the future detections they only exploit their previous known best classifier, behavior model of which may not be sufficient to capture behavioral feature of new samples.

While exploitation is good to maximize the QoE on one step, we also need to explore other classifiers not selected by greedy method to improve the estimated accuracy, because exploration may produce the greater total QoE in the long run. For example, if we identified $f_1$ is the classifier by greedy selection, while several other classifiers are estimated to be nearly as good but with uncertainty. The uncertainty is that there may exist one of these other classifier that is better than $f_1$ in future, but you don't know which one at time $t$. In our system design, we have to make the classifier selection on each time steps, then it may be better to explore other classifiers and discover which of them are better in the long run. We will present an algorithm in the next section (Section 4.3) to balance the trade-off between the exploitation and exploration of classifiers in order to have the highest expected QoE.

In on-line learning comparing the QoE against different classifiers isn't practical. A better measurement of the learning success is regret, which defined as following.

**Definition 2 (Regret for learning algorithm $\mathcal{A}$).** Given the total number of detection requests $T$ that cloud processed according to a on-line detection algorithm $\mathcal{A}$. The *cumulative regret* of $\mathcal{A}$ is the difference between the total QoE by applying the best classifier and the total QoE by following algorithm $\mathcal{A}$ in all $T$ detections. Accordingly, the *cumulative regret* is given by

$$ghh. \tag{4.3}$$

And the goal of the algorithm design for our bandit problem is to minimize the expected cumulative regret, which by linearity of expectation we have

$$\mathbb{E}\Bigg[Reg_{\mathcal{A}}(T)\Bigg] = T\mu^* - \mathbb{E}\Bigg[\sum_{t=1}^{T} Q^t(\mathcal{A})\Bigg]. \tag{4.4}$$

Where $\mu^* = \max_{1\leq i\leq K} \mu_i$ is expected QoE of the best classifier. we should note that there are theorems concerning lower bounds for expected cumulative regret. The algorithm we will introduce in Section 4.3 will guarantee $O(\sqrt{KT\log T})$. ■

### 4.2.2 Contextual Multi-Armed Bandit Framework

In previous multi-armed bandits framework, the QoE payoff is only determined by the classifier that selected at time $t$. Any side information about the sample is ignored. Notice the learning goal of cloud platform is to provide user the best QoE by choosing the best classifiers. To achieve the goal, cloud have to learn overtime which classifier perform the best for the next detection request. It is important to exploit the similarity of side information and take into account the information in future learning process. This problem can be naturally modeled as a contextual multi-armed bandit problem. That is the QoE of an detection not only depend on the selected classifier but also rely on how we explore the available contextual information to make the best learning. In this section we present the contextual multi-armed bandit formulation for our malware detection problem.

To include sample context in the setup, cloud platform will first extract the context features from the received client request and perform context clustering in order to choose a classifier according to the cluster result. We abstract the sample context feature at time $t$ using the notation $\theta^t \in \Theta$ with $\Theta$ being a $d$-dimensional feature space. The context can include information about various file properties of the sample such as file header and other standard file specification metadata. Note that contexts could also be features extracted from the sample rather than metadata information. Extracting specific features is more costly, so we will use only the side information as the context in this paper.

Under this contextual bandits formulation, we have a unknown distribution $P$ over $(\Theta, Q_\Theta(f_1), \ldots, Q_\Theta(f_K))$. On each round, a sample $(\theta, Q_\theta(f_1), \ldots, Q_\theta(f_K))$ is drawn from $P$, the context $\theta$ is announced, and then for precisely one classifier is chosen by a bandits algorithm, its instant QoE $Q_\theta(f_k)$ is revealed. A contextual bandits algorithm $\mathcal{B}$ choose a classifier at each time slot $t$, based on the previous observation sequence $\big\{(\theta^1, f_k^1, Q_{\theta^1}(f_k^1)), \ldots, (\theta^{t-1}, f_k^{t-1}, Q_{\theta^{t-1}}(f_k^{t-1}))\big\}$, and the current context $\theta^t$. The expected cumulative regret (learning loss) of algorithm $\mathcal{B}$ with respect to the oracle benchmark by time $T$ is given by

$$\mathbb{E}\Bigg[Reg_{\mathcal{B}}(T)\Bigg] = T\mu_\theta^* - \mathbb{E}\Bigg[\sum_{t=1}^{T} Q_\theta^t(\mathcal{B}(\theta))\Bigg] \tag{4.5}$$

The expected cumulative reward gives the convergence rate of the total expected reward of the learning algorithm to the value of the optimal solution. Any learning algorithm with sublinear regret $O(N^\gamma)$ (for $\gamma < 1$) will converge to the optimal solution in terms of the expected reward. In another words, the goal of algorithm $\mathcal{B}$ is to minimize its regret, which is equivalent to maximizing the total reward.

In the next section, we will propose an efficient learning algorithm that learns the *oracle* classifiers with sub-linear regret bounds. To enable rigorous regret analysis, we assume that if context information is similar, then the expected reward obtained by selecting the same classifier is also similar. Formally, we have Lipschitz condition

**Assumption:** A Lipschitz contextual multi-armed bandit problem is a pair of spaces – a contextual space $\Theta$ and a function $\mathcal{F}$. A instance of the problem is a function $\mu_{\theta\in\Theta} : \Theta\times\mathcal{F} \to [0, 1]$, which is Lipschitz in each coordinate, that is $\forall\theta, \theta' \in \Theta$, $\forall f_k \in \mathcal{F}$, there exists $M > 0$ and $0 < \alpha \leq 0$ such that

$$|\mu_\theta(f_k) - \mu_{\theta'}(f_k)| \leq M|\theta - \theta'|^\alpha \tag{4.6}$$

*Remarks:* The above assumption will hold if the expected QoE $\mu_\theta(f)$ is bounded $\forall\theta, f$ and $M$ is chosen sufficiently large and $\alpha$ sufficiently small. while we prefer smaller $M$ and $\alpha$ because in this case more similarity information can be exploited and may lead to a better system performance.

## 4.3 Contextual Bandits Learning Algorithm for QoE Optimization

In this section, we will discuss the details of context clustering and how we make use of the context information to optimize the expected QoE through the proposed contextual multi-armed bandit model. As discussed in the system model Section 4.2, the algorithm must balance exploitation and exploration to get good statistic performance. In the exploration phases, different classifiers are selected to learn their expected reward. In the exploitation phases, the classifier with the best estimated reward is selected in order to maximize the classification rewards. Note that the exploration and exploitation phases are interleaved unlike in the conventional learning approaches where only a single training phase is executed followed by the exploitation phase.

The expected QoE of different classifiers will differ because the length of behavioral feature collection have significant impact on the expected QoE $\mu(f_k^*)$ of a behavioral feature based malware classifier. Increasing the execution time will record more comprehensive behavioral features that generally lead to more accurate results. The improvement is mostly applicable to detect malwares that intentionally or unintentionally delay the malicious behavior after being analyzed. After all, short analysis will not capture any useful behavioral features for this type of malware and hence will lead to poor detection result.

### 4.3.1 Sample Context Feature Clustering

We observed there is exist some connections between the context information and the accuracy of the classifier $q(f_k)$, which in turn affects the expected QoE. For example, two groups of sample with significant different file properties may receive different QoE even though cloud system apply same selection policy. such as one group is packed software and the other group is non packed. The learning problem would be simple if there was no context information. But without using the context information the performance of the learning algorithm can be poor because the best oracle classifiers can be very different for different context information. Since the context space $\Theta$ can be very large and even continuous, learning the best oracle classifier for each individual context $\theta \in \Theta$ is extremely difficult, if not impossible. To overcome this obstacle, our learning algorithm will first partition the context space into smaller subspaces (i.e. context clusters) and learn the best oracle classifier within each subspace.

We take the K-means clustering algorithm as a context space partitioning subroutine in discussing our learning algorithm and it proofed effective in our experiment in Section 4.4. However, other clustering algorithm could also be implement to replace the K-means subroutine. The algorithm iterate through each training sample's context feature to assign the sample to the closest centroid in the metric of Euclidean distance, and recompute the mean of each centroid using the point assign to it. The K-means algorithm will always converge to some final set of means for the centroids. A partition of context feature space could be achieved by computing the Voronoi partition using the converged centroids.

Note that the converged solution may not always be ideal for our application and depends on the initial setting of the centroids. Therefore, in practice the K-means algorithm is run a few times with different random initializations. We choose the best centroids between different solutions by minimize the cost function

$$J(\ell^1, \ldots, \ell^T, \nu_1, \ldots, \nu_L) = \frac{1}{T}\sum_{t=1}^{T} ||\theta^t - \nu_{\ell^t}||^2 \tag{4.7}$$

where $\ell^t \in \mathcal{L} = \{1, \ldots, L\}$ is the index of cluster which the sample's context $\theta^t$ currently assigned to and $\nu_{\ell^t}$ is the context cluster centroids.

For a specific detection request, cloud extracts the received contextual features from client as the first step in the detection transaction. The extracted feature will be normalized for simplicity reason. For instance, if we decide to only include the file size as the context feature, the context space will be normalized with respect to the maximum file size and the minimum file size that cloud received so far. The normalized context features will be run through the pre-built clustering model and the cluster label of the input sample will be revealed. The learning algorithm will determine the optimal tracing length for this sample based on the context label and history rewards of the available classifiers.

Notice that for each specific sample with cluster label $y_\theta$, the realized QoE $Q_\theta(f_k)$ by selecting $f_k$ is an random variable drawn from an unknown distribution with mean $\mu_\theta(f_k)$, which is also initially unknown. However, we can estimate the expected QoE by observing many reward realizations from testing samples. Specifically, the best classifier under context $\theta$ is $f^*(\theta) := \arg\max_{f_k \in \mathcal{F}} \mu_\theta(f_k)$ and the best expected QoE for context cluster $y_\theta$ is $\mu_\theta^* := \mu_\theta(f^*(\theta))$. We call $f^*(\theta)$ the *oracle* classifier for context cluster $y_\theta$. The *oracle* classifiers are not know before hand by the on-line detection system but instead need to be learned. The learning is achieved by repeatedly test samples against classifiers of the cloud platform with a classifier selection policy $\pi$ that need to be designed.

### 4.3.2 Algorithm Description

Confidence bound is a standard statistics tool that commonly used to solve the exploitation and exploration trade-off in bandit problems. We propose an new algorithm in a similar vein with existing upper confidence bound (UCB) algorithms [76,77], but with context information and classifier updates. The formal description of the algorithm is presented in Algorithm 1 and we name it ConUCB. It uses sample context information to learn the best classifier for the context (thus the optimal dynamic analysis length) along the time horizon by maximize user's expected QoE of the malware detection service.

During the learning procedure, the algorithm maintains multiple counters and the estimated accuracy $\bar{q}_\ell(f_k)$ and the QoE $\bar{Q}_\ell(f_k)$ for each available classifier $\mathcal{F} = \{f_1, \ldots, f_k\}$ under different context type $\nu_\ell$. The counter $N_k^\ell$ records how many times the classifier $f_k$ has been chosen to classify samples whose context type is $\nu_\ell$ up to round $t$. The counter $N_k$ denote the total number of classifier $f_k$ being selected in all the $t$ rounds. The counter $N$ is the total number of samples that have been submitted to the cloud. In the bootstrap of the algorithm, each classifier is applied for every context type to initialize the estimated QoE $\bar{Q}(f_k)$. For each future samples

**Algorithm 1 ConUCB** Contextual Upper Confidence Bounds for Malware Detector Selection

```
Input: $\alpha \in \mathrm{R}^+$, $\mathcal{S} = \{(\boldsymbol{\theta}^1, \mathbf{x}^1), (\boldsymbol{\theta}^2, \mathbf{x}^2), \ldots, (\boldsymbol{\theta}^t, \mathbf{x}^t)\}$,
    $\mathcal{F} = \{f_1, f_2, \ldots, f_K\}$, $\mathcal{K} = \{1, \ldots, K\}$, $\beta \in [0, 1]$,
    $\mathcal{M} = \{\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_L\}$, $\mathcal{L} = \{1, \ldots, L\}$
Output: $\{y^1, \ldots, y^t\} \in \{0, 1\}$
Initialization:
for $\ell \in \mathcal{L}$ do
    for $k \in \mathcal{K}$ do
        Randomly select $(\boldsymbol{\theta}^m, \mathbf{x}^m)$
        Set $\bar{q}_\ell(f_k) \leftarrow f_k(\mathbf{x}^m)$
        Set $\bar{Q}_\ell(f_k) \leftarrow \bar{q}_\ell(f_k) - \beta c(\tau_k)$
        Set $N_k^\ell \leftarrow 1$
    end for
    Set $N^\ell \leftarrow K$,
end for
Set $N \leftarrow LK$,

for each malware detection request $(\boldsymbol{\theta}^t, \mathbf{x}^t)$ do
    $\ell^* = \arg\min_{\ell \in \mathcal{L}} ||\boldsymbol{\theta}^t - \boldsymbol{\nu}_\ell||^2$
    $k^* = \arg\max_{k \in \mathcal{K}} (\bar{Q}_{\ell^*}(f_k) + \sqrt{\frac{\alpha \ln N^{\ell^*}}{N_k^{\ell^*}}})$
    Set $r_t = f_{k^*}(\mathbf{x}^t)$
    Set $\bar{q}_{\ell^*}(f_{k^*}) \leftarrow \bar{q}_{\ell^*}(f_{k^*}) + \frac{1}{N_{k^*}^{\ell^*}} [r_t - \bar{q}_{\ell^*}(f_{k^*})]$
    Set $\bar{Q}_{\ell^*}(f_{k^*}) \leftarrow \bar{q}_{\ell^*}(f_{k^*}) - \beta c(\tau_{k^*})$
    Set $N^{\ell^*} \leftarrow N^{\ell^*} + 1$
    Set $N_{k^*}^{\ell^*} \leftarrow N_{k^*}^{\ell^*} + 1$
    Set $N \leftarrow N + 1$
end for
```

submitted, the algorithm first run the clustering routine to get the cluster type and then to select a classifier by taking into account both how close the current estimates are to be the maximum and the variance of the estimate. This could explore their potential for being optimal. After select the classifier and run the detection, the estimate of the QoE and the corresponding counters will be updated.

The quantity being maxed over in line 15 of the given algorithm is the upper confidence bound on the possible true QoE of the classifier $f_k$ for the particular context type, where the parameter $\alpha$ controls the width of the confidence interval. Each time a classifier $f_k$ is selected for context type $\nu_\ell$ the variance of $\bar{Q}_\ell^*$ is reduced because $N_k^{\ell*}$ is in the denominator of the variance term. On the other hand, each time a classifier other than $f_k$ is selected for context type $\nu_\ell$, the variance term of estimated QoE for $f_k$ will maintain unchanged. As time goes by it will be a longer wait, and thus a lower selection frequency, for classifier with a lower value estimate or that have already been selected more times for a particular context type.

In Algorithm 1, the exploitation and exploration phases are alternate implicitly in consecutive actions. If a classifier with large variance component (in the square root term) is chosen, we can view the action as explorative decision, since in such a case the upper bound is loose and taking $\bar{Q}_{\ell^*}$ as the estimate of the true expected reward is quite questionable. It is likely some other classifiers outperform $f_k$ in the measure of QoE. On the contrary, if an arm with large estimated QoE $\bar{Q}_{\ell^*}(f_k)$ is chosen, we can view the action as exploitative decision. Considering that $\sqrt{\frac{\alpha \ln N_k}{N_k^{\ell^*}}}$ decreases rapidly with each choice of $k$, the number of explorative decisions is limited. As $\sqrt{\frac{\alpha \ln N_k}{N_k^{\ell^*}}}$ becomes smaller, the average $\bar{Q}_{\ell^*}(f_k)$ gets closer to the true expected QoE $Q_{\ell^*}(f_k)$, and it is with high probability that the classifier corresponding to maximal QoE for the context type is indeed the optimal classifier for the context.

### 4.3.3 Learning Regret Analysis and Algorithm Complexity

In this subsection, we will first discuss the regret bound of the Algorithm 1 for contextual malware classification. The regret of the proposed algorithm depends on the sub-optimality of each non-optimal selection. We denote such a quality $\Delta_i = \mu_i - \mu^*, \forall i : \mu_i > \mu^*$. If there is no context information, we know from the classic work by [76] that the regret is bounded by $O(\sqrt{KT \log T})$. We can use this results to derive a regret bound for our proposed algorithm with context information. In the case with context, the regret bound will depends on the context clustering and the arrival process of different context types. Suppose in $T$ rounds we have total of $T_1, T_2, \ldots, T_\ell$ samples arrived that belongs to context type $1, 2, \ldots \ell$, respectively. So the regret of our algorithm is $R(\Delta_1, \ldots, \Delta_i) = \alpha_1\sqrt{KT_1 \log T_1} + \alpha_2\sqrt{KT_2 \log T_2} + \cdots + \alpha_\ell\sqrt{KT_\ell \log T_\ell}$, where the $\alpha$ is the constant coefficient. considering the worse case bound by maximizing the regret $R$ subject to $T = T_1 + T_2 + \cdots + T_\ell$. Solving this maximization problem give us the solution that $T_1 = T_2 = \ldots, = T_\ell = T/L$. Thus the regret bound for our ConUCB is $O(\sqrt{KLT \log T})$. Thus we can still obtain sublinear regret bound with context information, but penalized by an extra constant that related to the number of context clusters compared to the regret bound without context information in [76].

### 4.3.4 Contextual Bandits under User Interference

As the cloud platform provides user the optimal time length it needs to execute and classify the submitted sample, users might not alway accept the default time and may request a different value (Usually smaller because users are impatient) through the User Agent. To cope with this scenario, we have modified the Algorithm 1 to produce a modified version (Algorithm 2) called $\varepsilon$-ConUCB. The $\varepsilon$ stands for the probability of users refuse to accept the analytical time length yield by Algorithm 1.

**Algorithm 2** **$\varepsilon$-ConUCB** Contextual Upper Confidence Bounds for Malware Detector Selection with User Interference.

```
Input: $\alpha \in \mathrm{R}^+$, $\mathcal{S} = \{(\boldsymbol{\theta}^1, \mathbf{x}^1), (\boldsymbol{\theta}^2, \mathbf{x}^2), \ldots, (\boldsymbol{\theta}^t, \mathbf{x}^t)\}$,
    $\mathcal{F} = \{f_1, f_2, \ldots, f_K\}$, $\mathcal{K} = \{1, \ldots, K\}$, $\beta \in [0, 1]$,
    $\mathcal{M} = \{\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_L\}$, $\mathcal{L} = \{1, \ldots, L\}$
Output: $\{y^1, \ldots, y^t\} \in \{0, 1\}$
Initialization:
Initialize $\bar{Q}_\ell(f_k)$, $N^\ell, N^\ell_k,$, and $N$ as in ConUCB

for each malware detection request $(\boldsymbol{\theta}^t, \mathbf{x}^t)$ at time $t$ do
    $\ell^* = \arg\min_{\ell \in \mathcal{L}} ||\boldsymbol{\theta}^t - \boldsymbol{\nu}_\ell||^2$
    $k^* = \arg\max_{k \in \mathcal{K}} (\bar{Q}_{\ell^*}(f_k) + \sqrt{\frac{\alpha \ln N^{\ell^*}}{N^{\ell^*}_k}})$
    $k' \leftarrow$ User_Input($\varepsilon$)
    if $k' < k^*$ then
        Set $k^* \leftarrow k'$
    end if
    Update $\bar{Q}_{\ell^*}(f_{k^*})$, $N^{\ell^*}, N^{\ell^*}_{k^*},$, and $N$ as in ConUCB
    Set $r_t = f_{k^*}(\mathbf{x}^t)$
end for
```

As the user input in Algorithm 2 violated the specific routine employed by ConUCB to balance the exploration and exploitation, the performance of $\varepsilon$-ConUCB would suffer more regret compared to ConUCB. For each sample that submitted by user that doesn't accept the analytical waiting time, we assume the worst case regret is $\Delta$. Considering the probability of such detection task happening in the system is $\varepsilon$, the total regret for these non-compliant detection in T rounds is $\varepsilon T \Delta$. Similarly, the regret for the compliant detection tasks in T rounds is bounded by $O(\sqrt{(1-\varepsilon)KT\log(1-\varepsilon)T})$. Together we obtained the linear regret bound $O(\sqrt{(1-\varepsilon)KT\log(1-\varepsilon)T} + \varepsilon T) = O(T)$. This result is same as the regret bound of $\varepsilon$-greedy, therefore it may stuck on some suboptimal classifier. However, in the practical implementation we could always run ConUCB at the background while providing the specific no-compliant user with the suboptimal solution and maintain the overall regret sublinear.

## 4.4 Experiment Results

We implemented a prototype of the proposed system in an emulated laboratory environment and evaluated various aspects of its performance. In particular, we designed a set of experiments and used real-word malware samples collected from Internet to observe the dynamic actions in selecting the best available classifiers for each submitted samples based on the malware context features and accumulative QoE of each available classifier. We will show how our on-line learning algorithms presented in Section 4.3 learn to choose the best classifier given the sample context and achieve the optimized detection results.

Three major components of the system need to be deployed for sample submission from user agent, dynamic malware analysis virtual cluster at cloud, and system evaluation components. The user agent is implemented as a Chrome Extension, in which we have an utility component that built specially for the purpose of our experiment evaluation, removing the need for manual test. The utility component can automatically submit malware samples given a list of URIs of testing samples. All the samples will be submitted to cloud regardless of the projected detection time requirement. It is essentially a replacement of a human tester that repeatedly "click" URIs to download testing samples and respond to cloud proposed analysis length with designed actions.

The cloud continuously receives malware detection requests from different end-users through the browser Extension. In contextual detection mode (Algorithm 2), the *Context Feature Extraction* component first extracts the context information (such as various file meta data) associated with the submitted sample. The context information which we use in the experiments are the size of the executable, and the size of the PE code section (.text section) in the binary. Nevertheless, the framework can be applied to any context feature in general. For example, the packer information can also be added as a key context feature. In the samples we obtained, all the executables are non-packed Windows PE binary. After the context feature is clustered using K-means, the cluster label will be used to select a classifier to perform the malware detection. Once the classifier is selected, the analysis length will be determined correspondingly and a instrumented virtual machine will be deployed immediately to analyze the submitted sample for that long. After the dynamic analysis and feature preprocessing, the selected classifier will be used to predict the maliciousness of the sample under analysis. We now show the detailed results of our experiments.

### 4.4.1 Dataset and Context Clustering

The experiment dataset includes 3000 Portable Executable samples, among which 1500 are malicious and the other 1500 are benign software. The ground truth labels are obtained through Virus Total online scanner. We divide the sample set into three subsets with 1000 samples each. The first subset is for initial training, the second subset is for initial testing and continuous updating of the classifiers, and the third subset is for continuous testing. Figure 4.2 show the scatter plot of the context feature of the first two subset. We have selected the file size and the code section as context feature in our example not only because it is simple and intuitive but also because it is effective. The number of clusters is decided based on the metric of Silhouette score [78]. The score can help to identify clusters that are dense and well separated, which fulfills the requirements of context clustering. Table 4.1 show the calculated

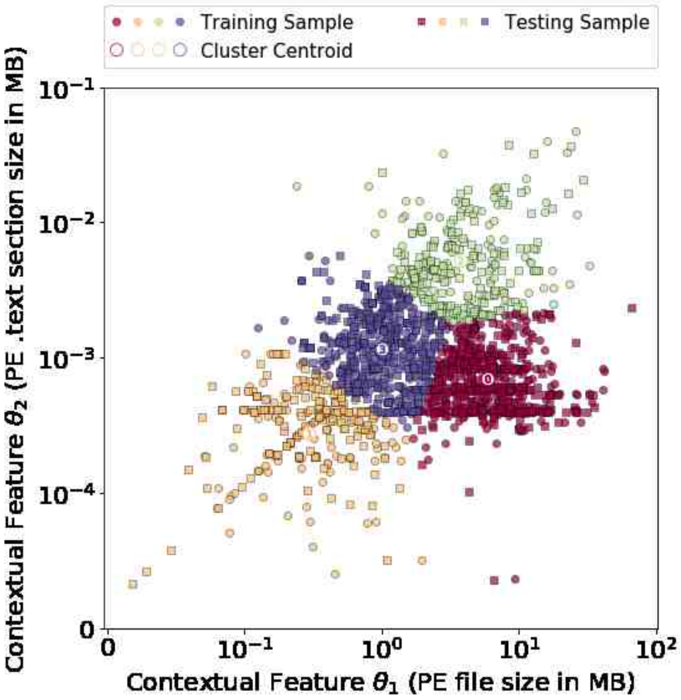


Figure 4.2: Context clustering without updating

value for different clusters. Due to the heterogeneous size distribution of our collected samples, We discovered it is better to use log scale for the context features clustering.

Table 4.1: Silhouette Coefficient for Number of Clusters

| Num. of Clusters | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| Silhouette Score | 0.368 | 0.408 | 0.446 | 0.415 | 0.392 | 0.368 | 0.358 |

### 4.4.2 The QoE and Classification Performance

In our experiment, we analyzed each training sample in our cloud detection system for 3 minutes and trained four individual classifiers using feature vectors extracted from profiles of the analysis for 0.5 minute, 1 minute, 2 minutes, and 3 minutes respectively. During the on-line learning process, selecting different classifiers to

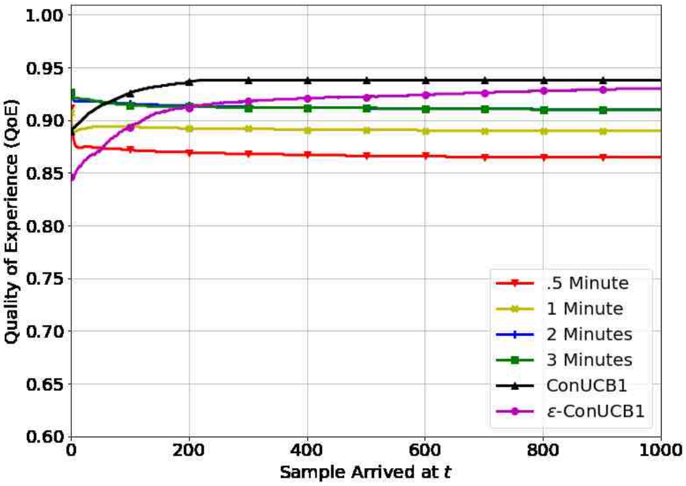


Figure 4.3: Normalized QoE comparison for $\beta = 0.01$ ($\varepsilon = 0.1$)

predict the testing sample will generate different QoEs as we defined in Section 4.2. For our performance evaluation, we take linear cost function i.e. $c(\tau_k) = \tau_k$ in the definition of QoE and compare the estimated expected QoE obtained by applying the proposed algorithms, namely ConUCB and $\varepsilon$-ConUCB, over the classifiers and the expected QoE obtained by each individual classifier.

The first experiment used the initial training set of 1000 samples to build the dynamic behavioral classifiers and conducted the evaluation using the initial testing set of 1000 samples. Figure 4.3 show the normalized accumulative QoE for $\beta = 0.01$. The QoE curves are obtained by calculating the expected value of 100 plays over the randomized testing sample sequences with the proposed on-line learning algorithm. The two proposed learning algorithms outperform all the individual classifiers. The ConUCB algorithm improved the maximum QoE of four individual classifiers from 91% to 94% after 1000 rounds of malware classification. The $\varepsilon$-ConUCB algorithm also gains up to 2% of rewards. Given that the experiment have moderate number of rounds and the context information used is limited to the size of the PE code section and the PE file size, a much higher performance gain can be expected when

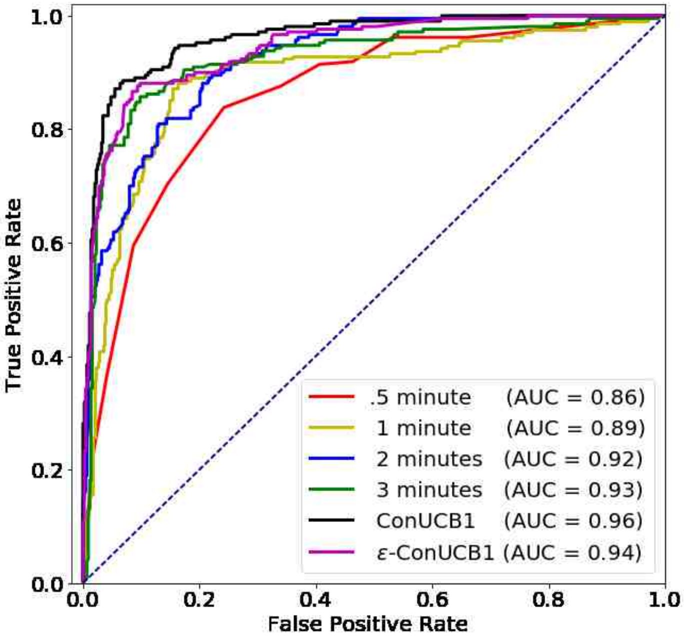


Figure 4.4: ROC curve and AUC comparison for $\beta = 0.01$ ($\varepsilon = 0.1$)

more rounds are played and more context information is available. In Figure 4.4, we compared the performance of the algorithms with performance of each individual classifiers, both ConUCB and $\varepsilon$-ConUCB achieved lower false positive rate than the individual classifiers. ConUCB and $\varepsilon$-ConUCB increased the area under the ROC curve to 96% and 94%.

In Figure 4.5, the normalized accumulative QoE is presented for $\beta = 0.1$. Compared to Figure 4.3, increasing the value of the cost coefficient $\beta$ will bring down the QoE of all the four basic classifiers, thus reduce the QoE of the ConUCB and $\varepsilon$-ConUCB because the algorithm tend to optimized towards the less accurate classifier that trained on 30 seconds of behavioral feature profiles. Figure 4.6 presents the corresponding performance comparison under the sample $\beta$. For $\beta = 0.1$, ConUCB increased the AUC by 1%, while $\varepsilon$-ConUCB have the similar performance as using single classifier that trained using 2 minutes of behavioral feature profiles.

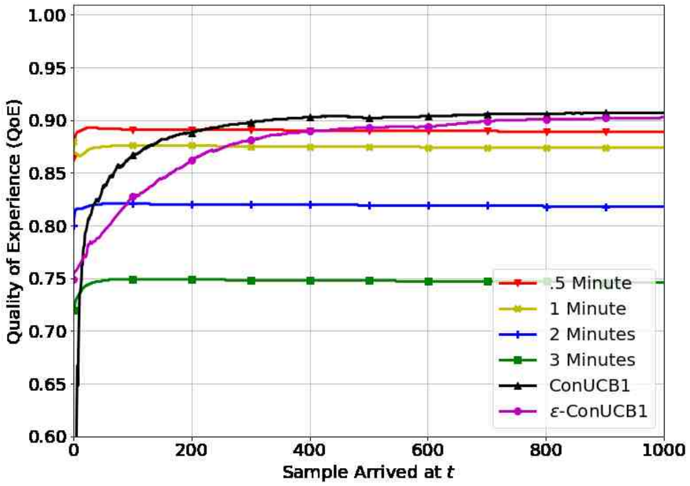


Figure 4.5: Normalized QoE comparison for $\beta = 0.1$ ($\varepsilon = 0.1$)

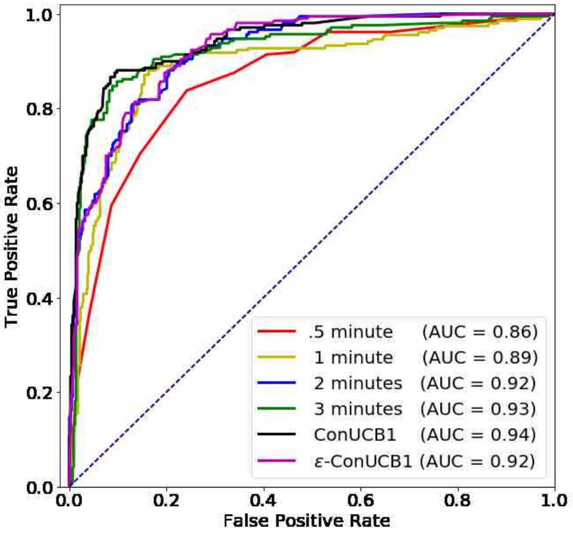


Figure 4.6: ROC curve and AUC comparison for $\beta = 0.1$ ($\varepsilon = 0.1$)

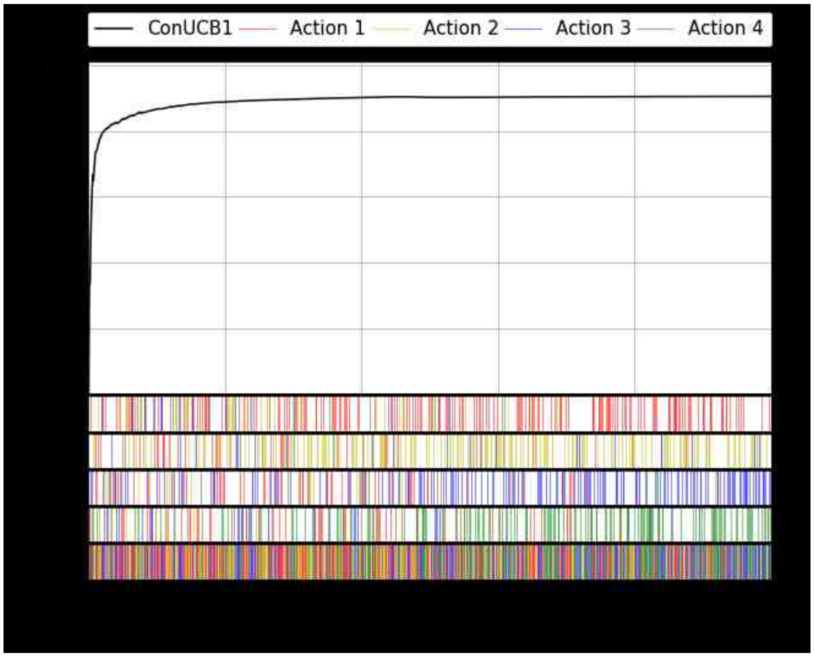


Figure 4.7: QoE and actions for each rounds

### 4.4.3 On-line Learning with Context Information

We have also studied each of the individual actions taken by algorithm ConUCB. Figure 4.7 display the classifier selection steps over a experiment of 1000 rounds and show the obtained QoE. The bottom color bar in the figure illustrates all the 1000 actions using four different colors, each of which represent an individual classifier that is selected in the step. The four color bars above it illustrate the actions taken under each different context cluster. Each row of these four color bar includes the action taken over samples belongs to a single context cluster in Figure 4.2. We can observed from the four color bars in the figure that each context cluster have gradually learned the best classifier to select under the particular context. For example, in the first 200 actions ConUCB has no preference on any particular classifiers and each classifier has the same probability of being selected. This corresponding to the exploration phase. On the other hand, when the play proceeds to the 800th round, the algorithm

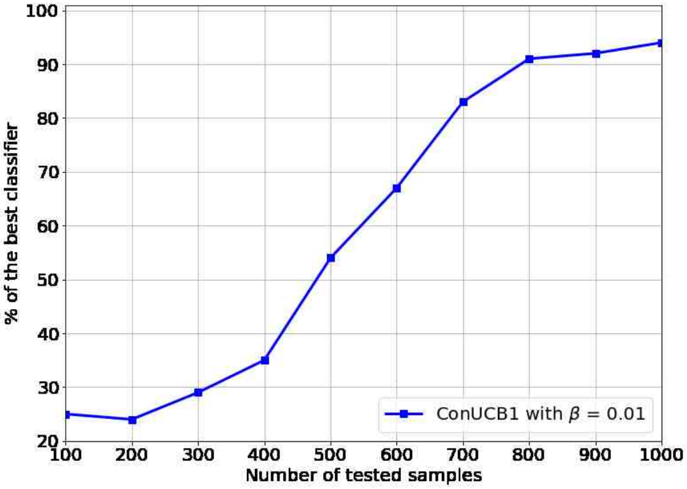


Figure 4.8: Percentage of the best classifier selected

is entering exploitation phase since the best classifier for the context is selected with high probability. Figure 4.8 show the percentage of the best action at different round by applying ConUCB with $\beta = 0.01$.

# CHAPTER 5

# Empirical Study of Static and Dynamic Features for Malware Detection

This chapter presents some of our preliminary study about the effectiveness of static and dynamic features using supervised learning approach in a collaborative environment. We will discuss how the different components in the system enable users to submit files to the platform for analysis and receive the malware detection decision. A detection decision could be a probability value indicating the maliciousness level of the sample or simply true or false value indicating a positive or negative detection. Users can submit executable files by a web interface or by the provided APIs. To sanitize the submitted data, the file manager has a duplication filter which checks the collection of submitted files for duplicates. The files passing through the filter are all new samples known to the platform and will be handled by the file manager. The file manager communicates with three core components: database, dynamic analysis sandbox, and malware classification engine. File manager is responsible for storing samples in the database, scheduling the dynamic execution of the samples in the sandbox, and running the classification algorithm. Result server and file analysis manager share the same components and positioned between the submission interface and the three system core components: malware detection model generator, database,

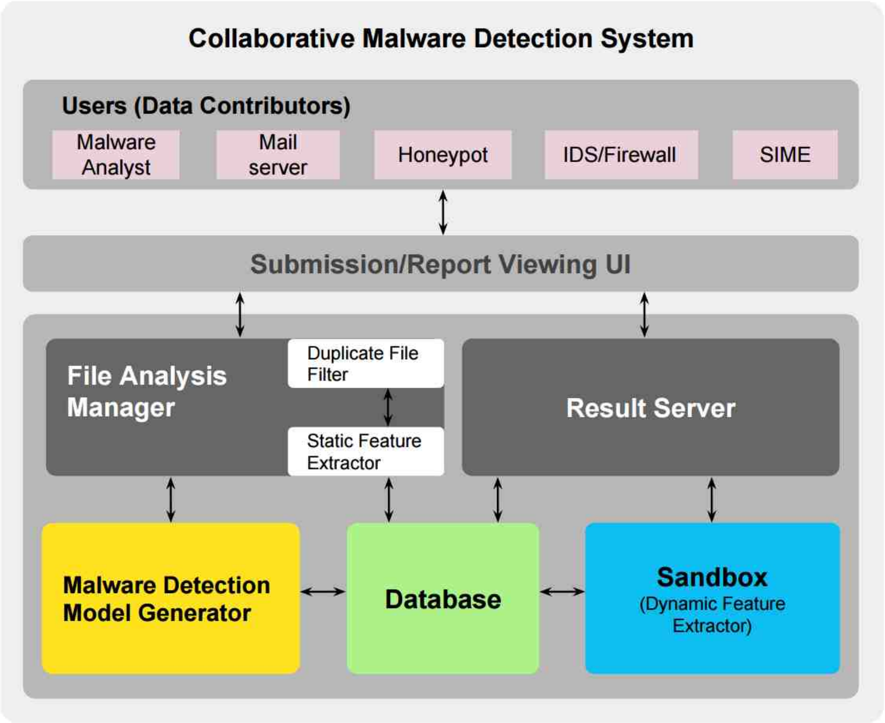

Collaborative Malware Detection System
Users (Data Contributors)
Malware Analyst
Mail server
Honeypot
IDS/Firewall
SIME
Submission/Report Viewing UI
File Analysis Manager
Duplicate File Filter
Static Feature Extractor
Result Server
Malware Detection Model Generator
Database
Sandbox
(Dynamic Feature Extractor)

Figure 5.1: System Architecture

and dynamic analysis sandbox. Figure 5.1 show the main components and data flows in the system. We will discuss each component separately.

## 5.1 System Components

### 5.1.1 Data Contributors

Since the system will be openly accessible, any entities that possess one or many malicious files are able to submit either by the provided submission APIs or by the system's web interface. Data contributor could be any or all of the following: malware analyst, mail server, honeypot, IDS/Firewall devices, SIEM, and private malware contributor. The contributor's capabilities are not limited to sharing malicious files. They are offered to access the detection engine that build on the collection of all contributed samples. In fact, being able to access the detection engine is the major motivation for them to sharing the malicious samples. In a word, the users are both the producer (malware data contributor) and the consumer (malware detection capability usage) in the system. For example, malware analysts who contribute samples may interest in using the platform to do initial scan of the samples and prioritize their manual analysis and forensic tasks. The willingness from private malware contributor may come from the fact that he/she discovered a malware campaign and want to raise public awareness of the threat. Contributions from mail server, honeynpot, IDS/Firewall, or SIEM system are mainly motivated by a more accurate detection engine they demand.

### 5.1.2 File Manger and Result Server

The platform accept malware submission both individually and collectively. File manager is the component that handle the uploading traffic, organize and queue the files to be processed in the next stage. It checks and filters duplicate submissions to prevent same file from being analyzed multiple times. Checking the file hash is an effective method to detect duplications. Each unique malware sample passing through the file manager will be sent to the static feature extractor. After extracting the static feature, file manager stores the malware sample, the static features of the sample, and various file identification information such as ssdeep hash and yara signature, into the database.

The result server has three key components. They are virtual machine manager (VMM), task scheduler, and report server. The VMM is responsible for scheduling, initiating, and terminating of virtual sandboxes that analyze the malware dynamically. Depending on system loads, our system can initialize multiple virtual sandboxes at the same time and run individual samples in parallel. This could greatly reduce the total time to analyze a large dataset by a factor of $n$, where $n$ is the number of VMs that run in parallel. Task scheduler distributes uploaded samples to the sandbox environment. In a production environment, various scheduler policy can be employed, such as user privilege based scheduling or threat severity level based scheduling. For example, sample submitted by prime users will be analyzed first. Newly discovered threats indicated by the file owner will be analyzed before the threats identified outdated. Report server relays the analysis data from the sandboxes to the database, it also responsible for forwarding the raw analysis reports to the use upon request. This raw analysis data can be rendered in a human friendly format for review by human analysis if necessary.

### 5.1.3 Database System

From the previous discuss, we learned the database can store the submitted samples, the static features, and the dynamic analysis reports. We also store history classifier object in the database. As our system keep evolving with more and more contributed malware samples, the current classification algorithm will be retrained and an improved classifier will be generated. The newly generated classifier will be upgraded in order to take advantages of the larger dataset. We use non-relational database for the data storage. On one hand, it is well suited for document storage and compatible with our Python APIs in retrieving feature set for the classification algorithms. On the other hand, non-relational database allow the malware storage system to scale horizontally to clusters of machines, which is necessary for our platform.

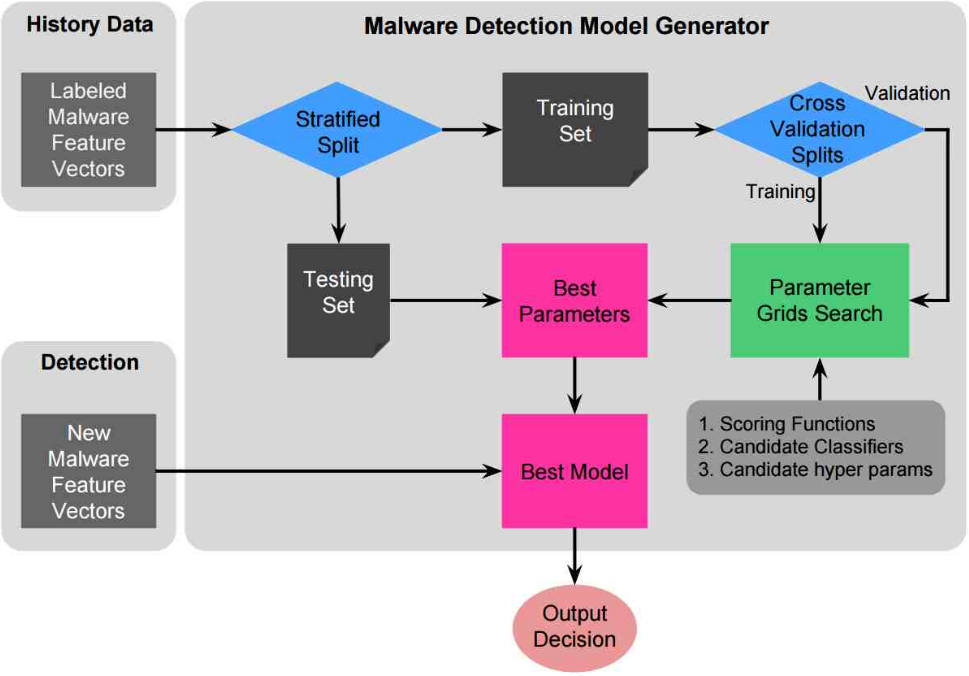


Figure 5.2: Architecture of Malware Detection Model Generator

### 5.1.4 Machine Learning Component

The machine learning component is the core of our proposed system. It is where all the contribution of this thesis originated from. The major innovation of this component is that it is capable of automatically improve the detection accuracy by running a serious of algorithms including stratified train and test data split, hyper-parameter search, and model selection algorithms. The flow diagram in Figure 5.2 shows the details of the machine learning component. At the input, It is up to the user to select the static features or the dynamic features to build the malware classifier. Once a user initiate the train procedure, the file manager will read the vectorized features from database and input to the machine learning component. The input data will be divided into two different sets, namely the training set and the testing set. The training set will be used in model searching for best hyper-parameters,

Table 5.1: Experiment Data Sets

| Data sets | Malicious | | Benign | |
|---|---|---|---|---|
| | Count | Size (GB) | Count | Size (GB) |
| Juniper | 1315 | 1.1 | 630 | 3.1 |

and the testing set is use in model selecting in order to obtain the best performance model. Since we apply supervised learning on only two classes: benign and malicious, it is very common that the number of samples in one class are far more than the number of samples in another class. To avoid issues cased by imbalanced dataset, we use straitified sampling method to divide the data into training and testing sets. The training data will be further split into two sets, one for training and another for validating in grid search, in order to find the best hyper-parameters for the classifiers. The classifiers with the best hyper-parameter set will be evaluated on the testing data, which have been left out for purpose of the model selection. At last, the best performed classifier on the training data is selected and applied as the detection engine for next period of system operation.

We formulate a public goods game model in the previous chapter based on signature detection. In this chapter, we present the procedure to build the proposed system and evaluate its performance using collective malware data. A game theoretic analysis of the system operation is constructed from the utility function obtained from experimental results. Specifically, we focus on implementation, evaluation, and operation of a malware detection system built on supervised learning classifiers for both static features and dynamic features. A hyper-parameter grid search algorithm is proposed to obtain the most accurate model fitting the cross validation data. The models obtained from both static features and dynamic features will be presented. We use various machine learning metrics to evaluate these best models. The purpose of these evaluations is to gain better understanding of the classification algorithm applied in the learning-based malware detection settings, thus improving its robustness in case of possible evasion attacks. Finally, we take a dataset composed of real malware samples captured in a corporate network. Subsets of the malware samples are submitted to the implemented system in order to simulate the public goods game. From the experimental results, we obtain a concrete utility function for the users, with which we achieved several very insightful findings. These findings will be presented throughout this chapter.

## 5.2 Experiments Results

We extracted both static and dynamic features for the testing dataset. The static feature was easy to extract via "pefile", a Python module to parse Portable Executable files. It can programmatically access most of the information contained in the PE headers as well as all sections' details and their data very efficiently. The dynamic feature extraction was more challenging. It would have been very difficult to build from scratch had we not discovered the software suite Cuckoo Sandbox. When properly configured, Cuckoo Sandbox can automatically initiate a pre-configured analysis guest, execute the sample in the isolated machine, and record all the operating system level activities as behavior features. Normally, Cuckoo Sandbox analysis reports contain useful dynamic behavioral features of the analyzed file. In our experiments, we used Cuckoo Sandbox to collect dynamic features for the classification task. Two virtual machines were set up to run the samples in parallel in a private cloud. The host computer where Cuckoo Suite was installed was a Debian Linux PC that has 8 cores, 16 GB RAM, and 3 TB disk drive. The machine learning software used in our experiments includes NumPy, SciPy, pandas, scikit-learn, and matplotlib.

Table 5.1 presents the overview of the dataset on which we experimented. The dataset includes a total of 1945 Win32 binary samples collected from the real world intrusion detection system of a corporate network during a one week period in 2015. Each sample in the dataset is labeled as either malicious or benign by the intrusion detection system. As shown in Table 5.1, the number of malicious samples is twice as much as the number of benign ones, while the size of the malicious set is only one third of the size of benign set. From these numbers we observe the fact that the average file size of a malware sample in this dataset is six time less than average size of a benign sample.

### 5.2.1 Static Features Extraction

We started our experiments by extracting the static features of the binaries because the overhead of static feature extraction was small but nonetheless has been proofed effective [13, 25]. The static features extracted from PE executables included all the PE header information and other meta data of the PE files such as symbol tables and the import tables. In our experiments, the total number of static file features extracted was 56. A complete set of static features used in our experiments is presented in Appendix A. All the static features were dumped using "pefile" tool. To build the model that fit the static feature well, our experiments ran a grid search algorithm over multiple classifiers with grids of hyper-parameter candidates.

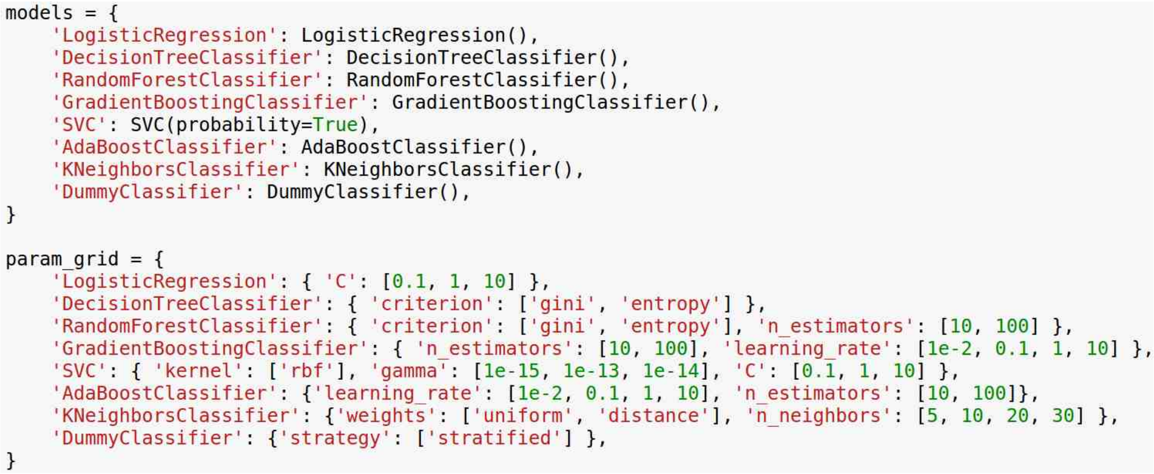
### 5.2.2 Hyper-Parameter Search Algorithm

Because we only possessed a limited experiment dataset, we had to use it carefully in order to produce valuable results. The algorithm used in conducting the grid search is presented in Algorithm 3. Generally, the grid search algorithm search for the highest scored classifier and corresponding hyper-parameters. It repeat the training and testing process on $k$ different stratified splits of the cross validation data, which is a subset of the overall dataset after set aside the testing samples. A average score of all the $k$ fold scores will be calculated and used to compare with different classifier's score. The classifier with highest average cross validation score will be selected. Briefly, using gird search algorithm to select the best model by evaluating various parameter settings can be seen as a way to use the labeled data to train the hyper-parameters.

**Algorithm 3** Grid search algorithm for the best model

```
Input: $S$: Dataset; $s$: score function; $H$: set of candidate classifiers; $G$: parameter
    space; $K$: set of stratified $k$ fold cross validation splits
Output: $H^{opt}$: best performance model evaluated on $S$
for each $clf \in H$ do
    for each parameter set $C \in G$ do
        for each CV split $cv(i) \in K$ do
            clf.fit(C, $(cv)_i^{train}$, $s$)
            cv_score(i) = clf.score($(cv)_i^{test}$)
        end for
        cv_mean = $\frac{1}{k}\sum^{i} cv\_score(i)$
        update the parameter $C$ for best performance of $clf$
    end for
    update the best performance model for the grid search algorithm
end for
```

The candidate classifiers and hyper-parameter grids are shown in Figure 5.3. The grid search algorithm is run independently for five different scoring functions: accuracy, precision, recall, F1, and receiver operating characteristic (ROC). With the illustration of Table 5.2, the first four evaluation metrics are defined as follows.



```
models = {
    'LogisticRegression': LogisticRegression(),
    'DecisionTreeClassifier': DecisionTreeClassifier(),
    'RandomForestClassifier': RandomForestClassifier(),
    'GradientBoostingClassifier': GradientBoostingClassifier(),
    'SVC': SVC(probability=True),
    'AdaBoostClassifier': AdaBoostClassifier(),
    'KNeighborsClassifier': KNeighborsClassifier(),
    'DummyClassifier': DummyClassifier(),
}

param_grid = {
    'LogisticRegression': { 'C': [0.1, 1, 10] },
    'DecisionTreeClassifier': { 'criterion': ['gini', 'entropy'] },
    'RandomForestClassifier': { 'criterion': ['gini', 'entropy'], 'n_estimators': [10, 100] },
    'GradientBoostingClassifier': { 'n_estimators': [10, 100], 'learning_rate': [1e-2, 0.1, 1, 10] },
    'SVC': { 'kernel': ['rbf'], 'gamma': [1e-15, 1e-13, 1e-14], 'C': [0.1, 1, 10] },
    'AdaBoostClassifier': {'learning_rate': [1e-2, 0.1, 1, 10], 'n_estimators': [10, 100]},
    'KNeighborsClassifier': {'weights': ['uniform', 'distance'], 'n_neighbors': [5, 10, 20, 30] },
    'DummyClassifier': {'strategy': ['stratified'] },
}
```

Figure 5.3: Candidate classifiers and hyper-parameter grids

Table 5.2: Table to illustrate evaluation metrics

| | Actual class | |
|---|---|---|
| Predicted class | TP (Correct result) | FP (Unexpected result) |
| | FN (Missing result) | TN (Correct result) |

$$
\begin{aligned}
\text{accuracy} &= \frac{TP + TN}{TP + FP + FN + TN} \\
\text{precision} &= \frac{TP}{TP + FP} \\
\text{recall} &= \frac{TP}{TP + FN} \\
\text{F1} &= 2\,\frac{\text{precision} \times \text{precision}}{\text{precision} + \text{precision}}
\end{aligned}
$$

Accuracy measures the fraction of all testing instances that are correctly classified; it is the ratio of the number of correct classifications to the total number of correct or incorrect classifications. For a malware detection system accuracy is one of the very important metrics to the users but is insufficient for our design purpose. Accuracy doesn't yield useful marginal interpretations, due to mixing of true positives and false positives instances. For example, in evaluating testing set with highly imbalanced positive and negative labels. A system with high precision but low recall returns very few true positive result, but most of its predicted labels are correct when compared to the training labels. A system with high recall but low precision is just the opposite, returning many true positive results, but most of its predicted labels are incorrect when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly. The F1 score is a weighted average of the precision and the recall, providing a unified metrics for compact evaluation results. It reaches its best value at 1 and worst score at 0.

The ROC curves typically feature true positive rate on the vertical axis, and false positive rate on the horizontal axis. This means that the top left corner of the plot is the "ideal point: a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

To be more specific, the series of best models are obtained through the following steps:

- Step 1: Run grid search algorithm to find the best parameter set for each of the candidate classifier. We obtain 8 different classifiers from this step.
- Step 2: Plot ROC curves and learning curves for the 8 classifiers obtained from previous step.
- Step 3: Choose the best classifier based on the ROC curve and learning curve. At the end of this step, we complete the model selection process with the best model at hand.

To evaluate the performance of different classifiers, we plotted ROC curves for the best model of each 8 different classifiers in Figure 5.4. To gain insight of how good the model performs, we plot the learning curve for each best model of the 8 different

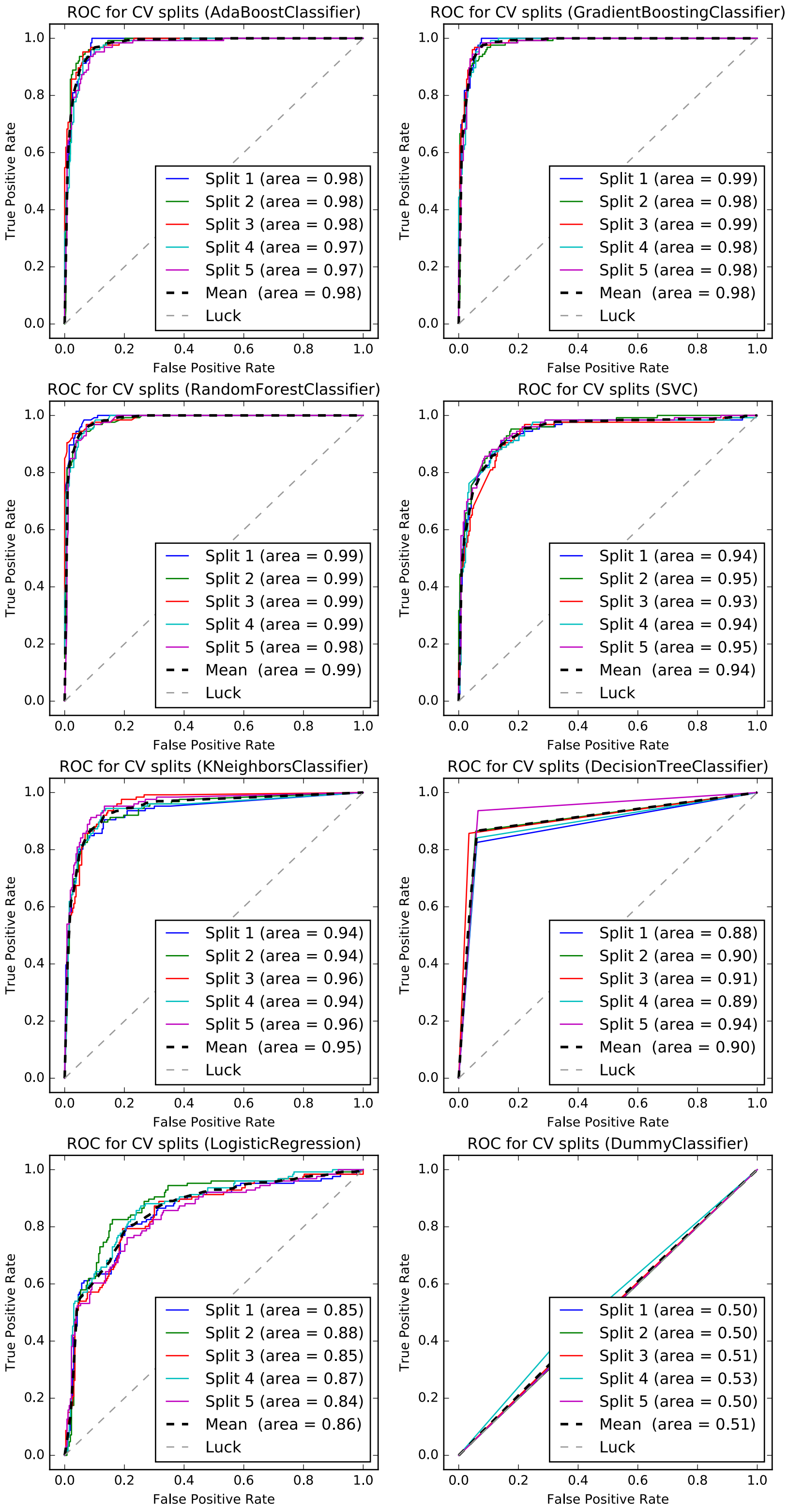


Figure 5.4: ROC curve for the best models returned by grid search for static features

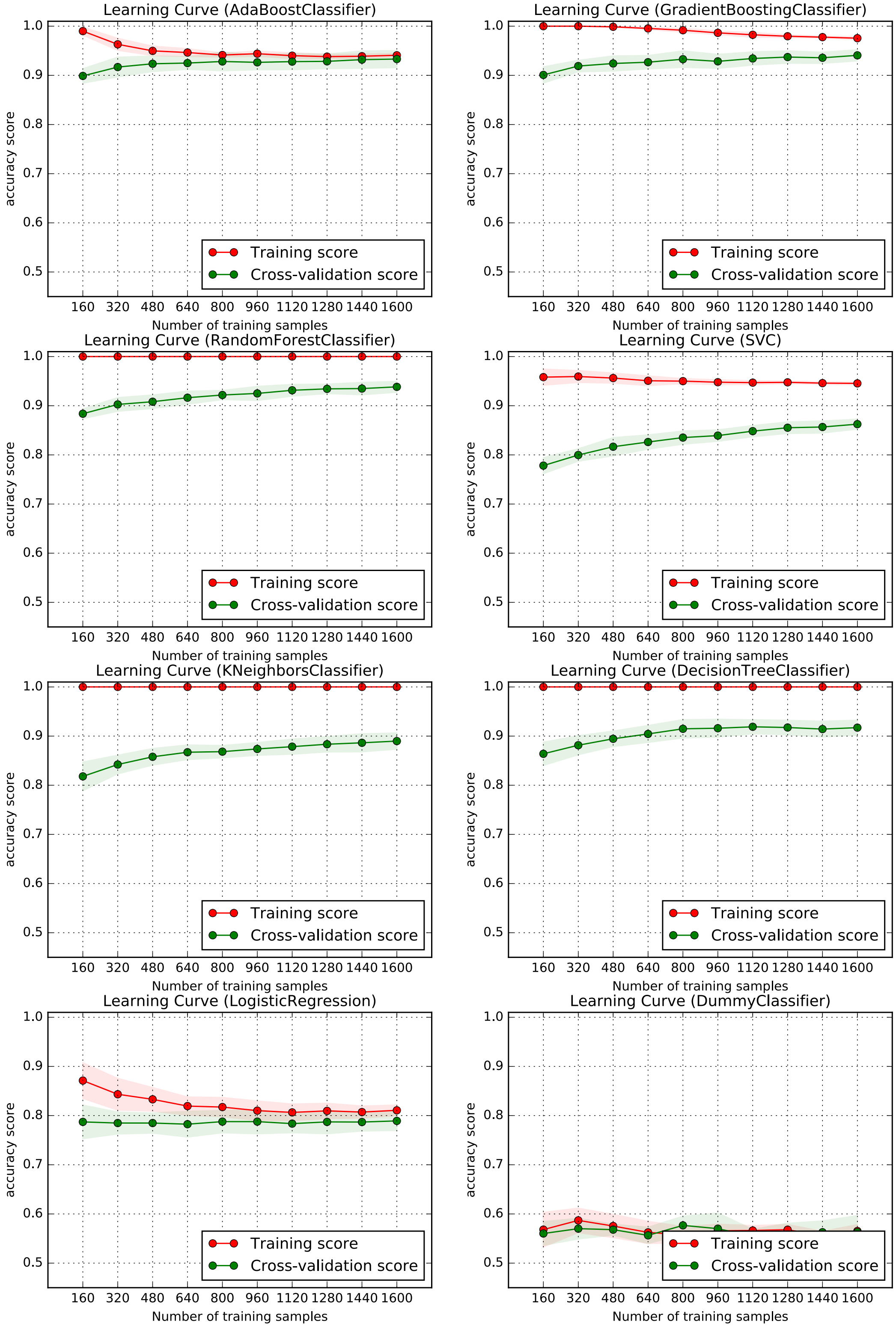


Figure 5.5: Learning curve for the best models returned by grid search for static features

classifiers. Figure 5.5 shows the learning curve. From the ROC curves and the calculated area under the curve (AUC), *AdaBoostClassifier*, *GradientBoostingClassifier* *RandomForestClassifier* stand out as very good classifiers for our classification problem. All the different folds of AUC for these three classifier are over 0.97. However, for *GradientBoostingClassifier* and *RandomForestClassifier*, all their AUCs are above 0.98. If we have to choose one superior classifier, *RandomForestClassifier* is the one to choose from the AUC's perspectives. However, *GradientBoostingClassifier*'s performance is very close to it. To get the classifiers performance from a different angle, we turn to the learning curve for a better evaluation and comparison of models. As we can see from the learning curve in Figure 5.5, *RandomForestClassifier* has high variance problem. The learning curve of *AdaBoostClassifier* is a ideal case when the training error and test error curve finally meet at a rate very close to 1. For a collaborative malware detection platform, we expect the detection accuracy will increase as the sample contribution increase. The *RandomForestClassifier* is perfect satisfy our requirement as its learning curve shows that the model improved its detection accuracy when the training data increases. It also have more space to improve following the trend presented. So we select the *RandomForestClassifier* as the classifier. As for all the other good model, they all inferior to the three classifiers discussed from both ROC curve and learning curve plots.

By comparing Figure 5.4 and Figure 5.5, we selected the *RandomForestClassifier* as our classifier for the game theoretic analysis. In the next, we will take the *RandomForestClassifier* model as our system detection model and conduct our experimental game analysis.

### 5.2.3 Dynamic Behavior Features and the Model

As we discussed in the Section 3.1, the detection system should based on features that are not easy to identify and mutate, so that evasion attacks by generating malware mutants are impossible. The following experiment method is based on dynamic behavior features. It provide an "adaptive" method that is able to continuously improve the detection model and therefore to achieve high accuracy, robust malware detection. We now present the procedures that achieved this objective.

### 5.2.4 Dynamic Behavior Features

The dynamic behavior features are generated from Cuckoo Sandbox. Juniper dataset was submitted to the latest version (2.0-rc1) of Cuckoo. In the feature engineering procedure, Cuckoo will be improved to capture more dynamic traits that identified important. For our first prototype, we use features from Cuckoo's vanilla report, which is the json formatted analysis result generated by out-of-box Cuckoo.

Table 5.3 is the settings used in vanilla cuckoo to collect our first version of behavior features. Cuckoo with two VMs takes 32.57 hours to execute all the files in the dataset. This time is very close to the theoretical value which is about $1945 \times 120/2 = 32.4$ hours. However, our study about execution status revealed the fact that no every sample executed exactly 120 seconds. As a matter of fact, 35 examples were not executed at all, some of the samples are terminated very early, others are executed more than 120 seconds. We also discovered that the average execution time for the benign set is 165 seconds, which is higher that the 102 average execution time for the malware set. It is noticeable that malware samples incur more frequent abnormal executions that cause it to terminate earlier. This complies with the fact that malware samples that evade Sandbox execution exist. Figure 5.6 is the

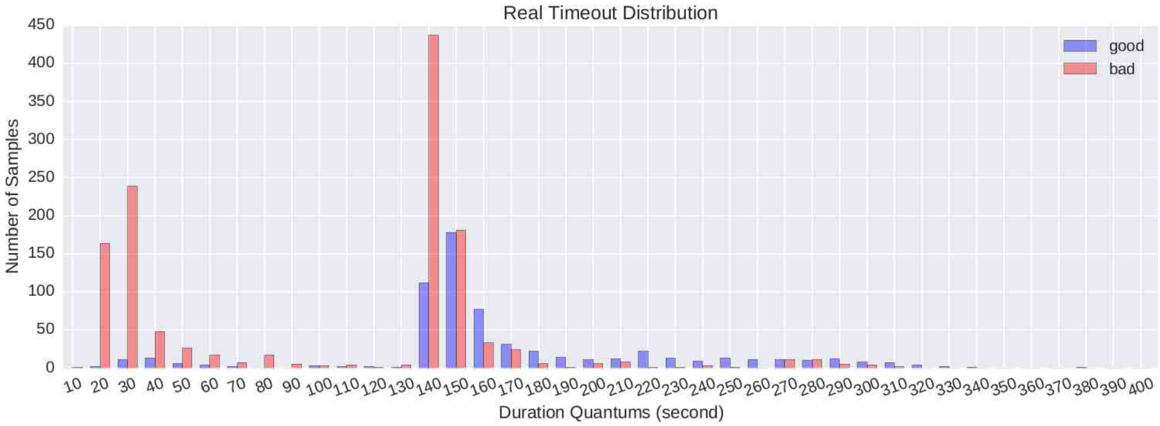


Figure 5.6: Execution duration quantums vs. Number of samples in the quantum

bar chart that show the real execution time for all the samples. The scatter plot of all the samples is presented in Figure 5.7, with duration of execution on x axis and file size on y axis.

Table 5.3: Summary of Cuckoo Sandbox settings I

| | **Options** | **Setting** |
|---|---|---|
| 1 | Analysis Timeout | 120 seconds |
| 2 | VMs in parallel | 2 |
| 3 | Network | Host-Only, InetSim |
| 4 | Volitality | Off |
| 5 | TCPDump | On |
| 6 | Screenshot | On |
| 7 | html reporting | On |
| 8 | json reporting | On |
| 9 | maongdb reporing | On |

### 5.2.5 Model Evaluation - Dynamic Features

The features we extracted is show in the Figure 5.8 with there respective importance score in the best model obtained from grid search. “duration”, namely the real execution time by Cuckoo is ranked the most important score in our best model. Another five most important features are “directory_created”, “file_exists”, “apistats”,

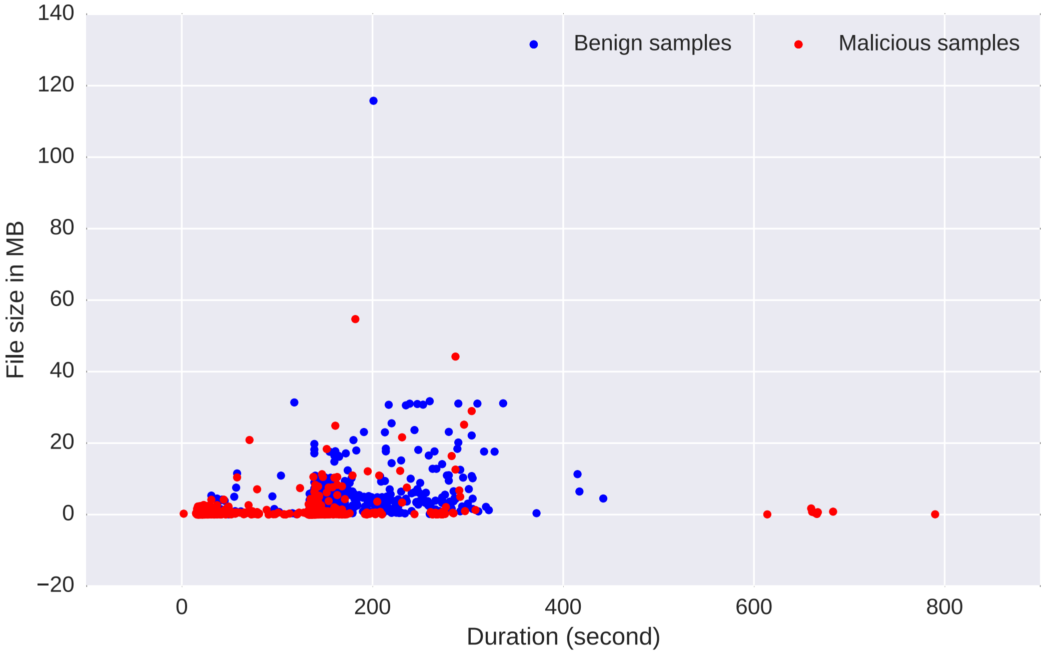


Figure 5.7: Scatter plot of duration and file size for the dataset

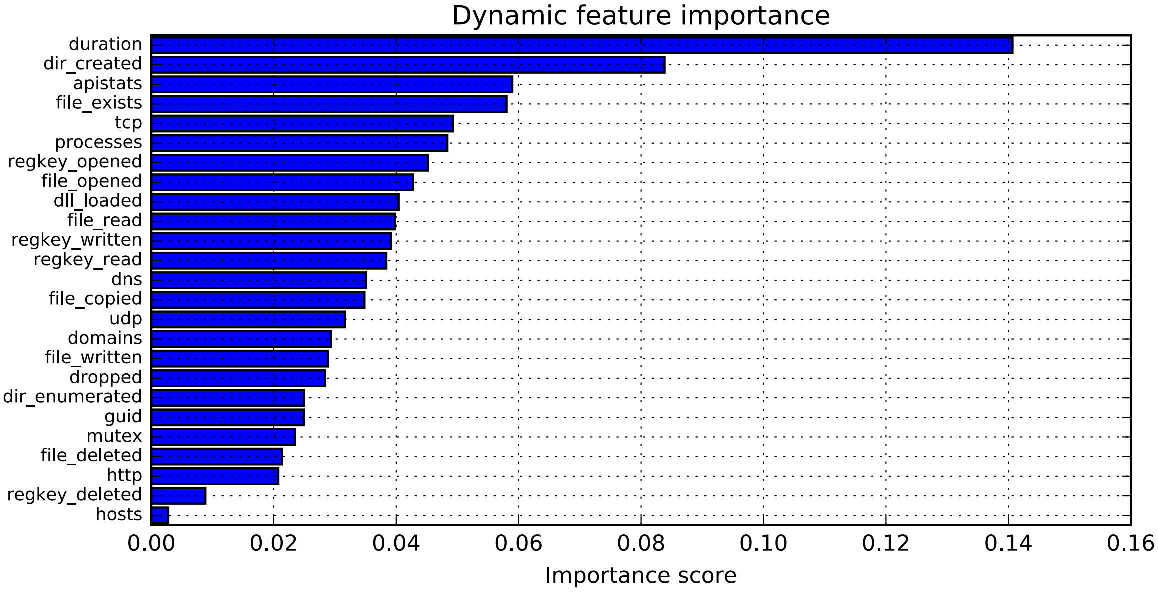


Figure 5.8: Feature importance ranking

“dns”, and “processes”. While the five least importance features are “udp”, “mutex”, “file_deleted”, “regkey_deleted”, and “hosts”. This ranking information is quite intuitive in general.

As mentioned above, grid search is used in obtaining a best model for dynamic feature. We employ the same grid search algorithm described in 5.2.2. Similarly, we used grid search algorithm to obtain the best hyper-parameters for each candidate classifiers. Once the best hyper-parameters for are found, we plot the ROC curves and learning curves for each individual classifiers.

We now present the performance evaluation of the models obtained from grid search for the dynamic behavior features. Figure 5.10 is the ROC plot of the grid search found classifiers. Figure 5.11 is the corresponding learning curves.

From Figure 5.10, we noticed that *RandomForestClassifier* excels in ROC and AUC. *GradientBoostingClassifier* and *AdaBoostingClassifier* are also very good candidates with respect to the metrics ROC and AUC for different folds of cross validation data. As we observed, all the other classifiers are low performance and we will not consider them for further discussion. We then turn to the learning curve in Figure 5.11 for further insights about the three ensemble classifiers that have good performance. The learning curves of these ensemble classifiers indicate they are potential good classifiers to select from because their testing scores are increasing with more samples included in the training set. However, the training scores for *RandomForestClassifier* and *GradientBoostingClassifier* are either remain highest (former) or keeping a gap between the corresponding testing scores (latter) for different training set sizes. These are all sign of overfitting. We further conclude that *AdaBoostingClassifier* is the best classifier of all three because its training scores are decreasing with growing number of samples and finally approaching the testing scores. The learning curve indicates

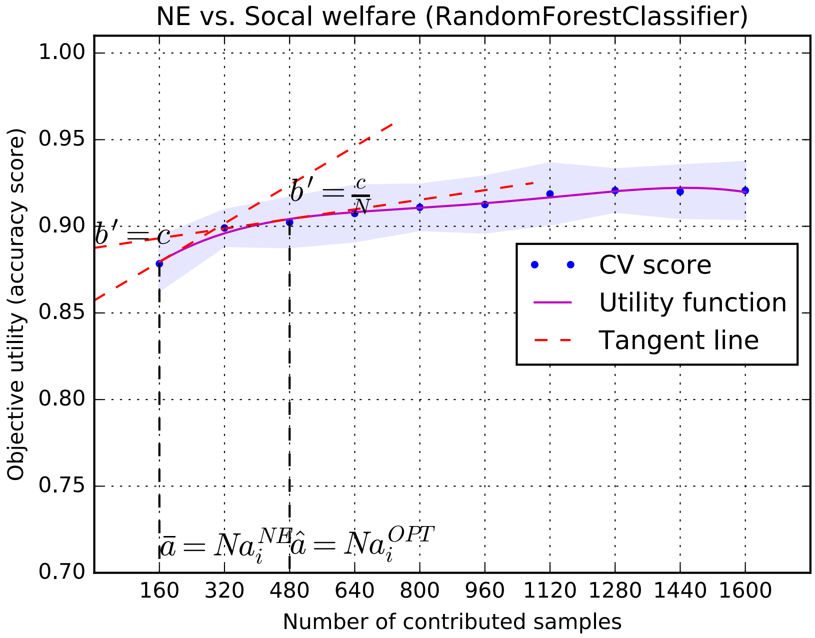


Figure 5.9: Utility function based on *RandomForestClassifier* and static feature

that the good performance of *AdaBoostingClassifier* is unlikely from overfitting the training data as it did in the other two ensemble classifiers. In this case, we choose the *AdaBoostingClassifier* as our detection engine.

Once the best model obtained, we can use numerical method to get the utility function. Figure 5.9 show the utility function of the machine learning based malware detection system based on dynamic feature vectors. For dynamic features we also plotted in Figure 5.12 the PoA when number of player is increased.

To compare with the result we get from static feature based classifier. We notice some differences in the PoA changes and the social optimal contribution changes with respect to the increasing of the players in the game. The PoA curve plotted based on static feature have increased smoothly, while the PoA curve for dynamic feature based model have experienced a leap at when the number of players in the game reached 37. This offer a great opportunity for the platform operator to keep the system work

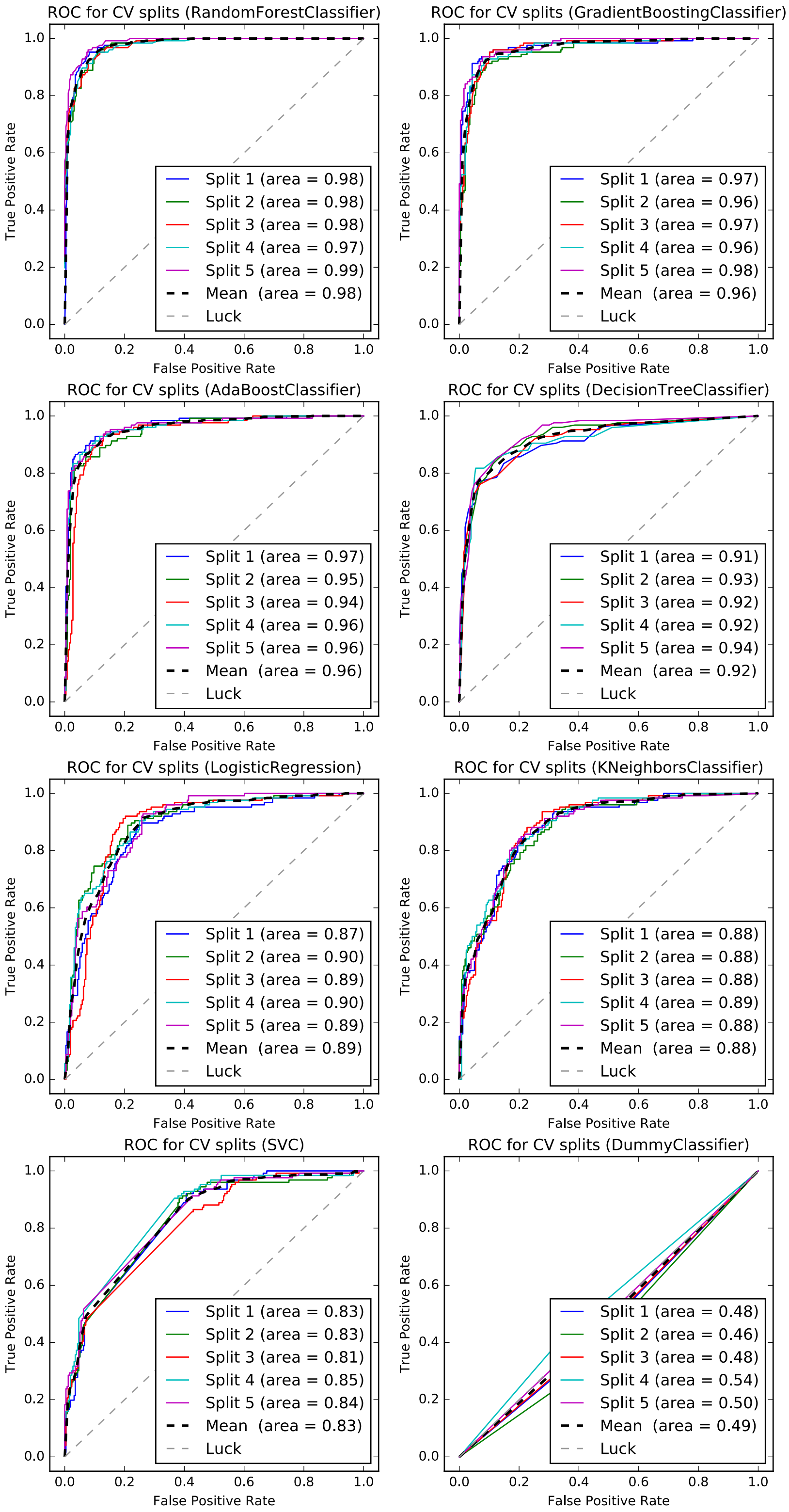


Figure 5.10: ROC curve for the best models returned by grid search for dynamic behavioral features

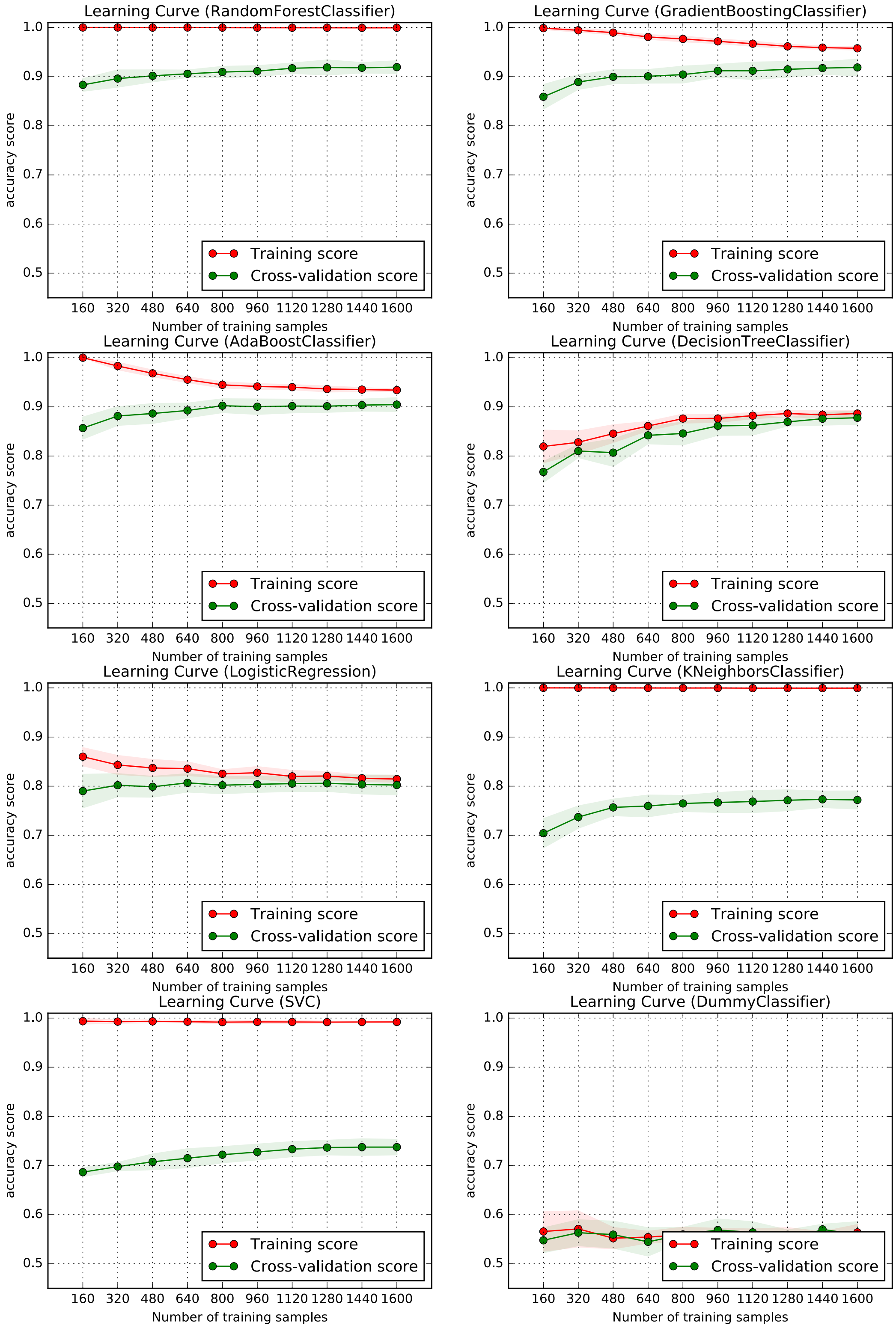


Figure 5.11: Learning curve for the best models returned by grid search for dynamic behavioral features

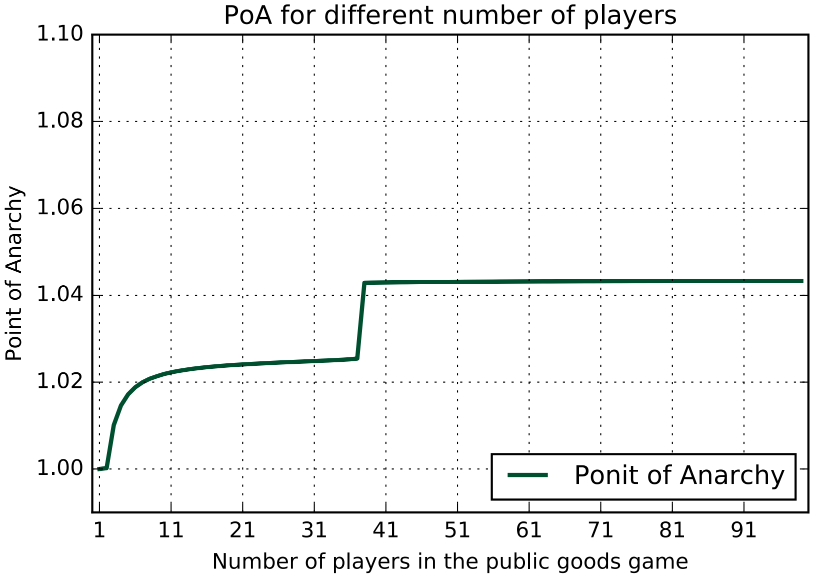


Figure 5.12: PoA obtained from experiments

in an efficient state by limiting the number of contributors. However, the concrete user throttle mechanism is heavily depends on how the operator monetize the utility.

## CHAPTER 6

# Conclusions

The number of malware threats increased tremendously in the past decade. The malware growth made malware based security breach becomes one of the major threat in the information security landscape. Because traditional signature-based malware detection method mainly relies on the work of human analysts to reverse engineer the malicious artifacts and create the signatures for detection. There are not enough security experts to handle the large set of malware samples, thus making the traditional signature-based method too costly to be effective. In this thesis, we proposed malware detection methods that leverage the effectiveness of large dataset analysis using machine learning on malware datasets.

## 6.1 Thesis Overview

We presented TroGuard, an efficient system solution to protect the client systems against web-based Trojan attacks that make use of social engineering techniques to convince victim users to willingly download and execute a legitimate-looking malicious software. TroGuard creates a profile dictionary of various *application functionalities (types)* through system-level activity observation and supervised classification of several applications of each category. TroGuard later employs the generated profiles across the client systems for online detection of the Trojan attacks that occur through website downloads. TroGuard notifies the user of his/her recent suspicious executable download if the functionality type of its system-level activity trace does not match the functionality type inferred by TroGuard's website analysis engine. Our results over a large set of applications show that TroGuard can effectively bridge the gap between the high-level user perceptions and low system-level execution traces in order to block Trojan intrusions efficiently under real-world adversarial situations.

To provide better user experiences through comprehensive dynamic analysis, TroGuard leveraged the advantages of cloud computing to analyze the malware samples in dynamically provisioned virtual machine sandboxes. This flexibility of dynamic analysis in cloud equipped TroGuard with the capability to balance the trade-off between the detection accuracy and the corresponding time and resource cost, thus providing the user with the best Quality of Experience for the given cloud infrastructure. To the best of our knowledge, this thesis is the first to define Quality of Experience as a performance metric for cloud-based malware detection. Chapter 4 proposed an on-line learning approach to select a particular classifier from a set of trained classifiers for each submitted file based on the static file feature of the sample and the classifiers history Quality of Experience measurements. The system model is based on contextual multi-armed bandit framework to balance the exploitation and exploration of the available classifiers. Our experiment results with 2000 real world samples demonstrated that the proposed framework effectively learned the best classifier based on the QoE measurement for a particular cluster of malware samples that share the same static file features. Accuracy study of the detection showed that the on-line learning approach also achieved better accuracy than each of the individual classifier.

## 6.2 Future Work

Although our work addresses many challenges in the design and modeling of practical dynamic malware detection system, the solutions we proposed have raised some new challenges. We conclude by discussing some open problems and directions for future research.

TroGuard provides the users with a practical security solution, one of our primary objectives in designing individual components in TroGuard has been to separate the low-level system details that are used for various analyses from the high-level information that is communicated to the user. As a case in point, while the kernel-level modules are tracing different system-level activities of the suspicious application, the user's interaction with TroGuard is through the browser extension using terms (potentially non-technical) that the user usually sees on popular download websites and is familiar with. In the simplest form, the user only has to confirm that the application class displayed by the browser extension matches the functionality he expected based on what he saw and read on the download page. Our usability study is conducted for a single subject, and we believe that such design points have made TroGuard usable by a vast range of users. However, there are many open questions remain. For example, what is the error rate of human in confirming the functionality proposed by website analysis? How frequently users abort the detection task after cloud have started the dynamic analysis? How these aborted tasks affect the overall performance of the platform. All those questions can be answered through an actual usability study in future.

TroGuard is built with various techniques to defend against adversaries users. The enhanced cloud version of TroGuard executes the proposed on-line bandits algorithm ConUCB to automatically select classifiers that are trained with different length of dynamic analysis based on static file features. Our experiment showed that the algorithm is able to progressively learn the optimal mapping from the clusters of static file feature (also refereed to as context type) of the tested samples to the available classifiers; However, we did not test the approach against new malware samples due to dataset limitation. As malware and its behaviors evolves across time, such evolution may affect the detection accuracy and is better to be taken into account when evaluating a practical malware detection system. Moreover, as the time progressed, the cloud platform is able to collect more and more samples, which could be used to improve the performance of each individual classifier. A possible solution to incorporate such dynamics into our system would be to enable system update with more training data and more accurate training labels that the system gained by learning. For example, the cloud detection system can trigger a system update after certain period of time or after certain number of processed samples. During the update, the classifiers could be upgraded with more training samples from recent detection; the static file feature space partitioning could also be improved by including static feature of newly detected malicious samples.

# APPENDIX A

# Static Features

Table A.1: Extracted static features vector by pefile

| No. | Feature | Value | No. | Feature | Value |
|---|---|---|---|---|---|
| 1 | number_of_import_symbols | 113 | 29 | major_version | 0 |
| 2 | size_code | 4096 | 30 | debug_size | 0 |
| 3 | sec_rawsize_rdata | 4096 | 31 | sec_rawptr_rdata | 8192 |
| 4 | pe_i386 | 1 | 32 | pe_char | 271 |
| 5 | sec_rawptr_text | 4096 | 33 | export_size | 0 |
| 6 | size_image | 20480 | 34 | sec_vasize_text | 3346 |
| 7 | iat_rva | 9256 | 35 | datadir_IMAGE_DIRECTORY_ENTRY_IMPORT_size | 100 |
| 8 | sec_rawptr_rsrc | 16384 | 36 | datadir_IMAGE_DIRECTORY_ENTRY_EXPORT_size | 0 |
| 9 | pe_minorlink | 0 | 37 | number_of_bound_import_symbols | 0 |
| 10 | sec_vasize_rdata | 2182 | 38 | datadir_IMAGE_DIRECTORY_ENTRY_BASERELOC_size | 0 |
| 11 | sec_entropy_rsrc | 1.02867676446 | 39 | pe_dll | 0 |
| 12 | sec_entropy_rdata | 3.20648735647 | 40 | compile_date | 1218437803 |
| 13 | minor_version | 0 | 41 | sec_rawsize_data | 4096 |
| 14 | sec_vasize_data | 468 | 42 | number_of_bound_imports | 0 |
| 15 | size_initdata | 12288 | 43 | sec_raw_execsize | 16384 |
| 16 | sec_rawptr_data | 12288 | 44 | sec_entropy_data | 0.442147583267 |
| 17 | sec_rawsize_rsrc | 4096 | 45 | pe_driver | 0 |
| 18 | pe_warnings | 0 | 46 | sec_entropy_text | 4.85296240301 |
| 19 | size_uninit | 0 | 47 | datadir_IMAGE_DIRECTORY_ENTRY_IAT_size | 468 |
| 20 | number_of_sections | 4 | 48 | sec_va_execsize | 7044 |
| 21 | generated_check_sum | 53913 | 49 | number_of_imports | 4 |
| 22 | std_section_names | 1 | 50 | virtual_size | 3346 |
| 23 | number_of_export_symbols | 0 | 51 | sec_rawsize_text | 4096 |
| 24 | pe_majorlink | 6 | 52 | datadir_IMAGE_DIRECTORY_ENTRY_RESOURCE_size | 1048 |
| 25 | check_sum | 0 | 53 | pe_exe | 1 |
| 26 | virtual_address | 4096 | 54 | number_of_rva_and_sizes | 16 |
| 27 | virtual_size_2 | 2182 | 55 | sec_vasize_rsrc | 1048 |
| 28 | total_size_pe | 20480 | 56 | sec_entropy_reloc | 0 |

Table A.2: Directly Mapped intermediate Level Attributes

| # | **Attributes** | **Value Type** | # | **Attributes** | **Value Type** |
|---|---|---|---|---|---|
| 1 | mail::filters | nominal | 41 | works-with-format::xml | nominal |
| 2 | mail::pop | nominal | 42 | works-with-format::zip | nominal |
| 3 | mail::smtp | nominal | 43 | works-with::3dmodel | nominal |
| 4 | protocol::bittorrent | nominal | 44 | works-with::archive | nominal |
| 5 | protocol::ftp | nominal | 45 | works-with::audio | nominal |
| 6 | protocol::http | nominal | 46 | works-with::im | nominal |
| 7 | protocol::imap | nominal | 47 | works-with::image | nominal |
| 8 | protocol::ip | nominal | 48 | works-with::mail | nominal |
| 9 | protocol::ipv6 | nominal | 49 | works-with::text | nominal |
| 10 | protocol::irc | nominal | 50 | works-with::video | nominal |
| 11 | protocol::jabber | nominal | 51 | interface::commandline | nominal |
| 12 | protocol::kerberos | nominal | 52 | interface::daemon | nominal |
| 13 | protocol::nntp | nominal | 53 | interface::shell | nominal |
| 14 | protocol::pop3 | nominal | 54 | interface::text-mode | nominal |
| 15 | protocol::smtp | nominal | 55 | interface::web | nominal |
| 16 | protocol::ssh | nominal | 56 | interface::x11 | nominal |
| 17 | protocol::ssl | nominal | 57 | x11::applet | nominal |
| 18 | protocol::tcp | nominal | 58 | x11::application | nominal |
| 19 | works-with-format::bib | nominal | 59 | use::browsing | nominal |
| 20 | works-with-format::docbook | nominal | 60 | use::chatting | nominal |
| 21 | works-with-format::dvi | nominal | 61 | use::compressing | nominal |
| 22 | works-with-format::gif | nominal | 62 | use::editing | nominal |
| 23 | works-with-format::html | nominal | 63 | use::filtering | nominal |
| 24 | works-with-format::info | nominal | 64 | use::gameplaying | nominal |
| 25 | works-with-format::jpg | nominal | 65 | use::learning | nominal |
| 26 | works-with-format::man | nominal | 66 | use::login | nominal |
| 27 | works-with-format::mp3 | nominal | 67 | use::monitor | nominal |
| 28 | works-with-format::oggtheora | nominal | 68 | use::organizing | nominal |
| 29 | works-with-format::oggvorbis | nominal | 69 | use::playing | nominal |
| 30 | works-with-format::pdf | nominal | 70 | use::printing | nominal |
| 31 | works-with-format::plaintext | nominal | 71 | use::proxying | nominal |
| 32 | works-with-format::png | nominal | 72 | use::scanning | nominal |
| 33 | works-with-format::po | nominal | 73 | use::searching | nominal |
| 34 | works-with-format::postscript | nominal | 74 | use::text-formatting | nominal |
| 35 | works-with-format::sgml | nominal | 75 | use::transmission | nominal |
| 36 | works-with-format::svg | nominal | | | |
| 37 | works-with-format::swf | nominal | | | |
| 38 | works-with-format::tar | nominal | | | |
| 39 | works-with-format::tex | nominal | | | |
| 40 | works-with-format::wav | nominal | | | |

Table A.3: Network Attributes

| # | **Attributes** | **Value Type** |
|---|---|---|
| 1 | Port(11380) | nominal |
| 2 | Port(12350) | nominal |
| 3 | Port(1863) | nominal |
| 4 | Port(40005) | nominal |
| 5 | Port(43277) | nominal |
| 6 | Port(443) | nominal |
| 7 | Port(49037) | nominal |
| 8 | Port(49038) | nominal |
| 9 | Port(5222) | nominal |
| 10 | Port(53) | nominal |
| 11 | Port(80) | nominal |
| 12 | Port(9997) | nominal |
| 13 | INET | numeric |
| 14 | INET6 | numeric |
| 15 | UNIX | numeric |
| 16 | NETLINK | numeric |
| 17 | Unique_IPs | numeric |
| 18 | Network_class_count | numeric |
| 19 | Total_recieve | numeric |
| 20 | Total_sent | numeric |

Table A.4: Resource Usage Attributes

| # | **Attributes** | **Value Type** |
|---|---|---|
| 1 | CPU_usage_Mean | numeric |
| 2 | CPU_usage_Stdev | numeric |
| 3 | CPU_usage_Virance | numeric |
| 4 | MEM_usage_Mean | numeric |
| 5 | MEM_usage_Stdev | numeric |
| 6 | MEM_usage_Virance | numeric |
| 7 | Memory_class_count | numeric |
| 8 | Process_control_class_count | numeric |

Table A.5: User Interactivity Attributes

| # | **Attributes** | **Value Type** |
|---|---|---|
| 1 | KB_Alpa | nominal |
| 2 | KB_Arrow | nominal |
| 3 | KB_Modi | nominal |
| 4 | KB_Num | nominal |
| 5 | KB_Punc | nominal |
| 6 | Mouse_Left | nominal |
| 7 | Mouse_Middle | nominal |
| 8 | Mouse_Right | nominal |
| 9 | Mouse_Wheel | nominal |

Table A.6: File System Attributes

| # | **Attributes** | **Value Type** |
|---|---|---|
| 1 | Dep(cc1plus) | nominal |
| 2 | Dep(codelite_indexe) | nominal |
| 3 | Dep(gdnc) | nominal |
| 4 | Dep(geniusreadline) | nominal |
| 5 | Dep(lame) | nominal |
| 6 | Dep(make) | nominal |
| 7 | Dep(perl) | nominal |
| 8 | Dep(pool) | nominal |
| 9 | Dep(StreamTrans_#5) | nominal |
| 10 | Dep(unitypanelser) | nominal |
| 11 | Dep(Xorg) | nominal |
| 12 | R(./..) | nominal |
| 13 | R(dev/..) | nominal |
| 14 | R(etc/..) | nominal |
| 15 | R(extcalc/..) | nominal |
| 16 | R(helloword.project/..) | nominal |
| 17 | R(home/..) | nominal |
| 18 | R(in/..) | nominal |
| 19 | R(initrd.img/..) | nominal |
| 20 | R(var/..) | nominal |
| 21 | R(lib/..) | nominal |
| 22 | R(opt/..) | nominal |
| 23 | R(proc/..) | nominal |
| 24 | R(run/..) | nominal |
| 25 | R(sys/..) | nominal |
| 26 | R(tmp/..) | nominal |
| 27 | R(ui/..) | nominal |
| 28 | R(usr/..) | nominal |
| 29 | W(/..) | nominal |
| 30 | W(npdf32Log/..) | nominal |
| 31 | W(dev/..) | nominal |
| 32 | W(etc/..) | nominal |
| 33 | W(extcalc/..) | nominal |
| 34 | W(helloword_wsp.mk/..) | nominal |
| 35 | W(home/..) | nominal |
| 36 | W(opt/..) | nominal |
| 37 | W(proc/..) | nominal |
| 38 | W(pwd/..) | nominal |
| 39 | W(tmp/..) | nominal |
| 40 | W(usr/..) | nominal |
| 41 | W(var/..) | nominal |
| 42 | File_access_class_count | numeric |
| 43 | Total_read | numeric |
| 44 | Total_write | numeric |

# Bibliography

[1] McAfee Antivirus Solution, "Top 10 malicious programs for Mac OS X; available at http://www.securelist.com," 2012.

[2] Webroot, "Insights from Collective Threat Intelligence," Tech. Rep. April, 2015.

[3] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis using Statistical Classification of Executables," 2008, pp. 301–310.

[4] S. M. Tabish, M. Z. Shafiq, and M. Farooq, "Malware Detection using Statistical Analysis of Byte-Level File Content," *CSI-KDD '09 Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pp. 23–31, 2009.

[5] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A Static, Packer-Agnostic Filter to Detect Similar Malware samples," vol. 7591 LNCS, pp. 102–122, 2013.

[6] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 255–264, 2002.

[7] A. Walenstein and M. Venable, "Exploiting Similarity Between Variants to Defeat Malware," *Proceedings of BlackHat Briefings DC 2007*, pp. 1–12, 2007.

[8] A. Karnik, S. Goswami, and R. Guha, "Detecting Obfuscated Viruses Using Cosine Similarity Analysis," *First Asia International Conference on Modelling & Simulation (AMS'07)*, pp. 165–170, 2007.

[9] M. Gheorghescu, "An Automated Virus Classification System," *Virus Bulletin Conference*, pp. 294–300, 2005.

[10] C. LeDoux and A. Lakhotia, "Malware and machine learning," in *Intelligent Methods for Cyber Warfare*, 2015.

[11] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale Malware Indexing Using Function-Call Graphs," *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[12] D. Kong, "Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification," *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1357–1365, 2013.

[13] C. Smutz and A. Stavrou, "Malicious PDF Detection Using Metadata and Structural Features," *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*, p. 239, 2012.

[14] N. Srndic and P. Laskov, "Detection of Malicious PDF Files Based on Hierarchical Document Structure," *Proceedings of the 20th Annual Network & Distributed Systems Symposium*, 2013.

[15] D. Maiorca and G. Giacinto, "Looking at the Bag is not Enough to Find the Bomb : An Evasion of Structural Methods for Malicious PDF Files Detection," *Proceedings of the ASIA CCS'13*, pp. 119–129, 2013.

[16] N. Srndic and P. Laskov, "Practical Evasion of A Learning-based Classifier: A case study," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 197–211, 2014.

[17] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers: A case study on pdf malware classifiers," *NDSS*, 2016.

[18] "VirusTotal," https://www.virustotal.com/, 2016, [Online; accessed March, 2016].

[19] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic Analysis of Malware Behavior using Machine Learning," pp. 1–30, 2011.

[20] U. Bayer, "Large-Scale Dynamic Malware Analysis," *PhD Thesis*, pp. 1–109, 2009.

[21] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable , Behavior-Based Malware Clustering," *NDSS*, pp. 51–88, 2009.

[22] U. Bayer, E. Kirda, and C. Kruegel, "Improving the Efficiency of Dynamic Malware Analysis," in *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, 2010, p. 1871.

[23] Google Safe Browsing, "Google Safe Browsing." [Online]. Available: https://safebrowsing.google.com/

[24] M. A. Rajab, L. Ballard, N. Lutz, P. Mavrommatis, and N. Provos, "CAMP: Content-Agnostic Malware Protection," *NDSS*, 2013.

[25] C. Smutz, "GMU-TR-2015-11 Discerning Machine Learning Degradation via Ensemble Classifier Mutual Agreement Analysis," George Mason University, Tech. Rep., 2015.

[26] P. Trinius, C. Willems, T. Holz, and K. Rieck, "A Malware Instruction Set for Behavior-Based Analysis," *Sicherheit Schutz und Zuverlässigkeit SICHERHEIT*, pp. 1–11, 2011.

[27] R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection," *2010 IEEE Symposium on Security and Privacy*, pp. 305–316, 2010.

[28] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015, pp. 11–20.

[29] M. Colajanni, D. Gozzi, and M. Marchetti, "Collaborative Architecture for Malware Detection and Analysis," *IFIP International Federation for Information Processing*, vol. 278, pp. 79–93, 2008.

[30] J. Oberheide, E. Cooke, and F. Jahanian, "CloudAV: N-version Antivirus in The Network Cloud," *Proceedings of the 17th conference on Security symposium*, pp. 91–106, 2008.

[31] L. Zeltser, "Free Automated Malware Analysis Sandboxes and Services," https://zeltser.com/automated-malware-analysis/, [Online; accessed March, 2016].

[32] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 557–566.

[33] Y. Okazaki, I. Sato, and S. Goto, "A new intrusion detection method based on process profiling," in *Applications and the Internet, 2002. (SAINT 2002). Proceedings. 2002 Symposium on*, 2002, pp. 82 –90.

[34] J. Zhang and R. J. Figueiredo, "Application classification through monitoring and learning of resource consumption patterns," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, 2006, pp. 144–144.

[35] J. Xu, L. Song, J. Y. Xu, G. J. Pottie, and M. van der Schaar, "Personalized active learning for activity classification using wireless wearable sensors," *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 5, pp. 865–876, Aug 2016.

[36] O. Atan, Y. Andreopoulos, C. Tekin, and M. van der Schaar, “Bandit framework for systematic learning in wireless video-based face recognition,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 9, no. 1, pp. 180–194, Feb 2015.

[37] Verizon, “2015 Data Breach Investigations Report,” *Information Security*, pp. 1–70, 2015.

[38] “Malware Information Sharing Platform,” http://www.misp-project.org/, 2016, [Online; accessed March, 2016].

[39] “Information Sharing Specifications for Cybersecurity,” https://www.us-cert.gov/Information-Sharing-Specifications-Cybersecurity, 2016, [Online; accessed March, 2016].

[40] “Malware Attribute Enumeration and Characterization,” http://maec.mitre.org/, 2016, [Online; accessed March, 2016].

[41] “AlienVault Open Threat Exchange,” https://www.alienvault.com/open-threat-exchange, 2016, [Online; accessed March, 2016].

[42] “ThreatConnect Collaborative Threat Intelligence Platform,” https://www.threatconnect.com/, 2016, [Online; accessed March, 2016].

[43] “IBM X-Force Exchange,” https://exchange.xforce.ibmcloud.com/, 2016, [Online; accessed March, 2016].

[44] “Facebook ThreatExchange,” https://developers.facebook.com/products/threat-exchange, 2016, [Online; accessed March, 2016].

[45] “Cyber Threat Alliance,” http://www.cyberthreatalliance.org/, 2016, [Online; accessed March, 2016].

[46] “Trojanhunter; available at www.trojanhunter.com,” 2013.

[47] R. Naraine, “Adobe: Beware of fake flash downloads; available at http://www.zdnet.com,” 2008.

[48] L. Garber, “Security, privacy, and policy roundup,” *IEEE Security & Privacy*, pp. 15–17, 2012.

[49] C. Kuo, F. Schneider, C. Jackson, D. Mountain, and T. Winograd, “Google safe browsing. project at google,” *Inc., June–August*, 2005.

[50] McAfee, “Mcafee antivirus solution; available at http://www.mcafee.com,” 2013.

[51] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, “Blade: an attack-agnostic approach for preventing drive-by malware infections,” in *Proceedings of the 17th ACM conference on Computer and communications Security*, ser. CCS ’10, 2010, pp. 440–450.

[52] P. J. Denning and R. D. Riehle, "The profession of it is software engineering engineering?" *Communications of the ACM*, vol. 52, no. 3, pp. 24–26, 2009.

[53] J. H. Saltzer and F. Kaashoek, *Principles of computer system design: an introduction*. Morgan Kaufmann Pub, 2009.

[54] J.-E. SNEDDON, "Malware hidden inside screensaver on gnome-look; available at http://www.omgubuntu.co.uk/2009/12/malware-found-in-screensaver-for-ubuntu," 2009.

[55] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *NDSS*. The Internet Society, 2009.

[56] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 5–14.

[57] D. Toupin, "Using tracing to diagnose or monitor systems," *Software, IEEE*, vol. 28, no. 1, pp. 87–91, 2011.

[58] E. Zini, "A cute introduction to debtags," in *Proceedings of the 5th annual Debian Conference*, 2005, pp. 59–74.

[59] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007, pp. 231–245.

[60] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the google chrome extension security architecture," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012, pp. 7–7.

[61] R. Smith, "An overview of the tesseract ocr engine," in *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*, 2007, pp. 629–633.

[62] Wikipedia, "Computer keyboard — Wikipedia, the free encyclopedia," 2013. [Online]. Available: en.wikipedia.org/wiki/Computer_keyboard\#Standard

[63] VMware, "VMware workstation," Published online at https://www.vmware.com/products/workstation/, 2013.

[64] G. Loiacono, F. Cecaro, and L. Vassallo, "Qube-OS," Published online at http://www.qube-os.com/, 2013.

[65] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, pp. 41–41.

[66] Bromium, "Understanding bromium micro-virtualization for security architects," Published online at http://www.bromium.com/sites/default/files/Bromium%20Microvirtualization%20for%20the%20Security%20Architect.pdf, 2013.

[67] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. ACM, 2011, pp. 285–296.

[68] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[69] S. Arlot and M. Lerasle, "$V$-fold Cross-Validation and $V$-Fold Penalization in Least-Squares Density Estimation," Oct. 2012, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00743931

[70] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, 2010, pp. 399–412.

[71] R. G. Anjoy and S. K. Chakraborty, "Efficiency of lttng as a kernel and userspace tracer on multicore environment," Tech. Rep., 2010.

[72] D. Maynor, *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research.* Syngress, 2007.

[73] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "Automated Malware Analysis." [Online]. Available: https://cuckoosandbox.org/

[74] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08, 2008, pp. 108–125.

[75] C. Smutz and A. Stavrou, "When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors," *NDSS*, pp. 21–24, 2016.

[76] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002.

[77] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A Contextual-Bandit Approach to Personalized News Article Recommendation," *WWW 2010*, p. 10, 2010.

[78] P. J. Rousseeuw, "Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, nov 1987.