University of Nevada, Reno

WOPR: A Dynamic Cybersecurity Detection and Response Framework

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

by
Aaron Walker

Dr. Shamik Sengupta/Dissertation Advisor

December 2021

THE GRADUATE SCHOOL

We recommend that the dissertation
prepared under our supervision by

Aaron Russell Walker

entitled

WOPR: A Dynamic Cybersecurity Detection and Response Framework

be accepted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

Shamik Sengupta, Ph.D.
Advisor

Sergiu Dascalu, Ph.D.
Committee Member

Haoting Shen, Ph.D.
Committee Member

Shahriar Badsha, Ph.D.
Committee Member

Hanif Livani, Ph.D.
Graduate School Representative

David W. Zeh, Ph.D., Dean
Graduate School

December, 2021

Abstract

Malware authors develop software to exploit the flaws in any platform and application which suffers a vulnerability in its defenses, be it through unpatched known attack vectors or zero-day attacks for which there is no current solution. It is the responsibility of cybersecurity personnel to monitor, detect, respond to and protect against such incidents that could affect their organization. Unfortunately, the low number of skilled, available cybersecurity professionals in the job market means that many positions go unfilled and cybersecurity threats are unknowingly allowed to negatively affect many enterprises.

The demand for a greater cybersecurity posture has led several organizations to develop automated threat analysis tools which can be operated by less-skilled information security analysts and response teams. However, the diverse needs and organizational factors of most businesses presents a challenge for a "one size fits all" cybersecurity solution. Organizations in different industries may not have the same regulatory and standards compliance concerns due to processing different forms and classifications of data. As a result, many common security solutions are ill equipped to accurately model cybersecurity threats as they relate to each unique organization.

We propose WOPR, a framework for automated static and dynamic analysis of software to identify malware threats, classify the nature of those threats, and deliver an appropriate automated incident response. Additionally, WOPR provides the end user the ability to adjust threat models to fit the risks relevant to an organization, allowing for bespoke automated cybersecurity threat management. Finally, WOPR presents a departure from traditional signature-based detection found in anti-virus and intrusion detection systems through learning system-level behavior and matching system calls with malicious behavior.

For Amanda

Acknowledgements

I would like to express my gratitude toward my advisor, Dr. Shamik Sengupta, for his guidance and support for my research activities. I would also like to thank my committee: Dr. Sergiu Dascalu, Dr. Haoting Shen, Dr. Shahriar Badsha, and Dr. Hanif Livani.

Furthermore, I would like to acknowledge the support of Research & Innovation and the Cyberinfrastructure Team in the Office of Information Technology at the University of Nevada, Reno for facilitation and access to the Pronghorn High-Performance Computing Cluster.

I thank my research partners who have aided and challenged me in so many ways which helped lead me to this milestone. Finally, I thank my family for their tremendous support through difficult times.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, we discuss the following:

- A brief introduction to malware threat assessment
- The motivating problem
- Limitations of existing work and discuss briefly how the contributions of this dissertation address these limitations.

1.1 Malware threat assessment

Malware, i.e., malicious software, presents a threat to computing environments in the office, at home, and in travel. Malware artists develop software to exploit the flaws in any platform and application which suffers a vulnerability in its defenses, be it through unpatched known attack vectors or zero-day attacks for which there is no current solution. Malware which successfully exploits such a vulnerability produces an information security incident. Security incidents may involve the loss of functionality of a system, compromise of user account credentials, unauthorized access into a system, or any number of conditions which compromise the confidentiality, integrity, or availability of a system or application.

Malware threat assessment involves the identification and evaluation of risks associated with the potential actions of a malware agent[1]. This involves understanding the differences between a vulnerability, a threat, and a risk. A vulnerability is a flaw in software or system design which can be exploited by a threat, resulting in the loss of confidentiality, integrity, or availability. The cost of this loss is considered the risk associated with a given threat.

The objective of a malware threat assessment is to provide recommendations for actions which limit risk and, in particular, what cybersecurity incident response strategies would be the most beneficial for an organization.

1.2 The motivating problem

Current methodologies regarding cybersecurity risk assessment tend to focus on the identification and remediation of software and system vulnerabilities, as seen in the popular Common Vulnerability Scoring System (CVSS)[2]. While we agree that the assessment of vulnerabilities is a key component of a robust cybersecurity program, we feel that this is only one side of the coin when considering risk. Malware acts as an agent to exploit vulnerabilities and therefore analysis of malware threats should be performed to gain a fuller picture of what risks present the greatest danger to an organization.

However, as shown in the following chapters, malware comes in many forms. There are several malware families and variants of malware within these families. The number of potential malware threats any organization may face makes human analysis of malware extremely difficult for organizations who find difficulty in placing and retaining key cybersecurity talent[3]. This led us to perform research into developing novel means of automating malware analysis using machine learning and producing threat intelligence information in the form of malware family classification. These new techniques of malware family fingerprinting from operating system application programming interface (API) function calls observed through behavioral analysis of malware as well as API sequences provide a means of automated malware analysis which can detect both known and previously unknown (zero day) malware threats. In so doing, our goal was to show that bespoke automated malware threat analysis and risk response is more accurate and beneficial to an organization than generalized automated solutions. Our approach was to create a framework titled WOPR, a name inspired by the fictional supercomputer known as War Operation Plan Response (WOPR) which was programmed to predict threats leading to nuclear war and execute automated responses in the 1983 movie WarGames[4]. Our WOPR is meant to predict malware threats and automate malware risk response using software behavioral analysis and machine learning.

1.3 Limitations of the State-of-the-Art

Current approaches have the following limitations:

- Current methods of analyzing Application Programming Interface (API) system calls made by malware do not consider which API calls are relevant to particular malware families.
- Current implementations of machine learning (ML) for malware analysis require expert ML knowledge in order to train effective models and therefore are not accessible for the majority of users.
- Many ML malware analysis methodologies only separate benign and malicious software and not which malware family the malware belongs to.
- Current methodologies of malware threat assessment rely on general-purpose rubrics. If an organization is aware that certain types of malware have a more damaging effect on their business, this will have an effect on the prioritization of incident response.

1.4 Contributions

Accomplished contributions, directly related to this dissertation are listed below:

- We show that the analysis of Windows API system function call frequencies observed through the behavioral analysis of malicious and benign software provides machine learning models which have higher real-world accuracy due to an understanding of the differences between malicious and benign behaviors.
- A machine learning methodology for identifying key traits of a malware family based on the relationship between antivirus signature and API call behavior.
- An examination of Windows API system function calls, observed through behavioral analysis of malicious and benign software, provides machine learning models which have higher real-world accuracy due to an understanding of the differences between malicious and benign behaviors, as well as differences between malware families.
- Real experimental malware data is used by collecting sequences of behavioral system API function calls. This allows for the fingerprinting of malware families.
- This research investigates the efficiency of Extreme Learning Machines (ELM) for malware family classification through learning API call sequences. Our experiments show that ELM is much faster than traditional ML methods and supports real-time classification of malware.

- We provide WOPR, a novel, bespoke malware threat and risk evaluation framework to assist in the prioritization of malware remediation which overcomes the limitations of existing “one size fits all” approaches to measuring malware criticality.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows.

- Chapter 2 describes a background study on malware analysis and presents some of the open research issues.
- Chapter 3 describes the flaws in the malware threat score provided by Cuckoo Sandbox.
- Chapter 4 analyzes linear and non-linear machine learning algorithms and compares their relative accuracies against data sets with different proportions of malicious and benign software.
- Chapter 5 shows how behavioral analysis of malware combined with a data set of Windows API function call frequencies classified by anti-virus signature matches allows for the creation of machine learning models capable of identifying the appropriate family to which a given malicious software belongs.

- Chapter 6 presents a framework for quickly learning the differences between software to accurately predict, not only if a given software is malicious or benign, but also to classify malicious software by family type. This approach requires the observation of only the first 3,000 API calls, rather than the full sequence of malware behavior.

- Chapter 7 shows that Extreme Learning Machines (ELM) can be used to accurately predict malware family classification through analysis of Windows API call sequences with the benefit of significantly reduced prediction time. This approach allows for near real-time analysis of malware through the use of Online Sequential Extreme Learning Machines (OS-ELM).

- Chapter 8 identifies flaws associated with preconfigured threat assessment systems and methodologies, with particular emphasis on the applicability of the resulting malware criticality score within the context of individual organizations and presents a novel bespoke malware risk classification framework.

- Chapter 9 discusses the conclusions of the given research and puts forward the future research ideas.

Chapter 2

Related Work and Research Challenges

In this chapter we describe the current state of the art in terms of cyber risk assessment, malware analysis, threat detection and response, as well as the current tools used in the attempt to provide solutions to these problems. Limitations and insights will be discussed as well as the novel forward direction our research provides. Summarizing, the main contributions of this chapter are as follows.

- We discuss the application of machine learning to the identification of malicious behavior in software.
- We explore the role of malware analysis in threat management.

- We explore the current methods and drawbacks of cyber security risk management.
- The challenges for cyber risk management are discussed.
- We describe the novelty of our research.

2.1 Identifying Malicious Behavior via Machine Learning

Machine learning provides a means of identifying malware by training a machine learning algorithm to learn the difference between malicious and benign software behavior. A user-centric machine learning framework [5] has been shown to be effective at identifying malicious behavior. This methodology involves establishing a baseline of 'normal' behavior through statistical analysis and then training a Multi-layer Neural Network with this baseline behavior. Any actions observed by the user are then compared to this model, with malicious behavior observed with a mean of 80.7% success. These results are promising, but the drawback of this methodology is in its ability to scale with a large organization of many hundreds or thousands of users. A challenge exists in relating the risk faced by the organization when different users may have vastly different threat profiles. If no two employees necessarily have the same baseline 'normal' behavior, an additional methodology of understanding user risk in relation to organizational risk is necessary for prioritization of security incident response.

Research into the sequence of Windows API system calls has allowed for malware classification[6]. This methodology is of interest to our research as in the current work we describe API frequency for malware classification. Though this research shows that API sequence can be used for fingerprinting malware, it also shows the need to identify API system calls which are relevant to particular malware families.

Additionally, research has been performed to consider the effects of anti-virus labels on malware detection, focusing on the machine-language opcodes discovered during the run time of malicious software[7]. This study shows that there is a negative correlation between AV labels and the ability to distinguish malware and benign software. While the results of this research are promising for accurate detection via opcode analysis, we believe that AV labels work well for API frequency analysis given the relationship between known malware families and their specific behavior.

The evaluation of API function calls as well as character strings in the malware code have been used as a predictor of future malware family variants[8]. The methodologies in this research include extraction of semantic features, incremental clustering of variants into families, and deep neural networks to learn patterns of malware evolution. We decided to borrow directly from the malware family classification provided by Microsoft Defender Antivirus to label malware family rather than this method, partly for ease of use but also to avoid any complications associated with defining our own malware family definition scheme.

2.1.1 Machine Learning Application for Cyber Risk Assessment

Machine learning is widely used in the field of cybersecurity and there are a number of different machine learning algorithms available for research[9], including decision trees and logistic regressions, to name but a few. Static analysis of malware involves inspection of the code at rest and has been shown to be successful in the classification of malware family [10]. This includes the examination of the register, operation codes, Portable Executable structured information and more. Dynamic analysis has proven effective in cases where static analysis would fail due to encryption or dynamic code loading [11], making this approach more attractive for the extraction of features for machine learning.

Recent research into bench marking machine learning models for network security [12] has demonstrated differences between deep learning and classical shallow learning methods in regard to detecting malicious network traffic. Various machine learning algorithms have been shown to be effective in cyber security, especially in relation to malware analysis, intrusion detection, and phishing email analysis [13]. One challenge for organizations wishing to implement machine learning models lies in the expert knowledge required to train effective models.

Bayesian network-based cyber security risk assessment has been shown to be more effective when machine learning algorithms are used to learn the parameters of the risk assessment model from historical data and online observations [14]. This methodology provides greater objectivity than other methods which rely solely on expert knowledge. This is particularly interesting when considering response to unknown or zero-day attacks, as these present threats which a subject matter expert may not be able to associate with an organizational risk in a timely manner.

An interesting (and currently unsolved) question regarding the practical use of machine learning for cyber security lies in how an organization should implement such techniques. A lack of expert knowledge with regard to machine learning is certainly a barrier for bespoke applications developed within an organization, which in turn leads to third-party "black box" solution acquisition. If an organization is unable to train such a solution within the context of that organization in terms of baseline "normal" behavior with the ability to dynamically adjust over time, new and unknown cyber threats may go unnoticed.

The use of machine learning in identifying cyber threats via time series classification has been shown in [15] where an SVM based approach was used in a framework for real-time monitoring of processes on embedded cyber-physical systems (CPS) devices. Deep learning techniques for time series classification are reviewed in [16] with multiple practical applications. Identification of Android malware via API data sets is performed in [17], [18], and [19] using LSTM to classify a given software as malicious or benign with very high accuracy. However, this methodology only provides a binary classification and does not detect to which malware family does the application belongs to.

The cost of processing overhead must be considered when evaluating methods for malware detection. Probability scoring has been shown to be effective in using a CNN based methodology for comparing gray scale images of known malware. As shown in [20], the cost of malware detection can be reduced by performing only this static analysis when a known malware is detected based on a probabilistic analysis, relegating dynamic analysis for only when static analysis alone is inefficient.

2.2 Malware Analysis for Threat Management

Malware remains a threat to organizations, be it in the form of targeted delivery through email or malicious web advertising, just to name a few attack vectors [21]. Threat actions can be observed from malware through the use of a malware sandbox [22], which we describe in Chapter 3. In addition to simply detecting malware, the observation of the effects of the malware on a system provides data regarding what threats the malware present. This dynamic analysis has been used extensively in the following research.

Dynamic analysis typically involves the use of a sandbox environment, such as Cuckoo Sandbox[22] which reports behaviors in terms of system API calls. Smith et al.[23]

demonstrated the potential for understanding malicious API calls through machine learning algorithms. The large number of Windows API calls found in malicious as well as benign software samples frustrates the process of feature selection for machine learning algorithms. One approach is to categorize the function calls based on their general function[24][25] and then evaluate the entropy of these categorial functions based on Information Gain[26], which essentially is a measure of how much information a randomly chosen data element in a set will teach us about another randomly chosen element in a set. Heuristic N-Grams analysis also adopts the Information Gain technique and has been shown to be effective in distinguishing malware from benign software[27]. This shows that while both benign and malicious software perform many of the same Windows API calls, their relative frequencies are distinguishable.

Another approach for selecting which Windows API calls to use as features involves narrowing the scope of analyzed malware samples to model specific malware families, such as WannaCry ransomware[28]. Malware family classification can be enhanced with machine learning models, as shown in[25], however here we also see the same issues with feature extraction and definition. Malware authors are aware of the attempts of researchers and system defenders to identify malicious software and often employ anti-analysis features[29]. As a result there has been research into image processing with Deep Learning – in this way, machine learning has been used in malware classification based upon image processing using an extracted local binary pattern[30]. There is no currently defined methodology for accurate, non-biased feature extraction of malicious behavior observed through dynamic analysis; therefore it is not reasonable to assume that bias is restricted without a systematic approach for the comparison of machine learning models with differing data sets.

2.2.1 Malware Relationships

Previous research has shown how dynamic analysis of malware through the use of the Cuckoo Malware Sandbox[22] can be used to detect malware and identify malware families within a limited scope[31]. We decided that it should be possible to perform a similar analysis on a much larger dataset, focusing on the classification of malware family rather than more generally on detecting malware from a mixed set of malicious and benign software.

Malware behavioral analysis involves the examination of system API function calls to determine malicious behavior. This provides information regarding such actions as Windows registry changes, file creation and deletion, process injection, and many other activities. One drawback of most solutions lies in the individual examination of these calls without context of process inter-dependencies. Recent research has shown that sentiment analysis offers a technique which may consider the relationship between API function calls of behavioral sequences [32]. By examining these relationships, the overall function or purpose of the malware may be inferred.

Another malware detection methodology based upon behavioral analysis involves examining the similarity between various API function calls [33]. This methodology involves computing the Levenshtein distance [34] between API calls to determine their similarity. Though this measurement is grammatical in nature, this lends well to the naming conventions of Windows API system function calls. By itself, this does not seem to be a methodology which is superior at detecting malware in comparison to previously described methods. However, the inferences achieved may help to identify relationships between malware as well as malicious activity in the general sense.

When malware is detected, another problem lies in automatic classification of malware family. To reduce the number of false positives, it is shown that using a combination of static and dynamic analysis is more beneficial in identifying and classifying malware than using a single feature such as a signature [17]. Even machine learning techniques can be thwarted by a malware author and lead to misclassification through adversarial examples - this has led to analysis of malware through image analysis performed within generative adversarial networks [35].

Interesting research into inferring malware families through the use of phylogenic networks makes use of Windows API calls to compare malware samples [36]. This approach is successful in drawing relationships within known malware families via behavioral analysis. While successful in revealing internal relationships among known malware families, it would be interesting to extend this research into discovering relationships between unlabeled malware samples to determine if real family identification is possible through this method.

A natural language processing (NLP) approach for malware classification from API sequences [37] has been shown to be effective when partnered with machine learning algorithms such as support vector machines (SVM), K-nearest neighbor (KNN), random forest (RF), and multilayer perceptron (MLP) when splitting API call sequences into n-grams. One reason why we may want to classify malware lies in threat response. If an organization is aware that certain types of malware have a more damaging effect on their business, this will have an effect on the prioritization of incident response.

Malware visualization has been used for static malware classification via both convolutional neural networks[38] and the intriguing bag-of-visual-words methodology with roots in natural language processing[9]. These visualization techniques have shown to be very effective for detection malicious software. However, since the visualization is generally performed upon the raw binary file, the presence of obfuscated or encrypted code may interfere with the detection process, thus limiting the efficacy of these methodologies. The use of dynamic methods for analyzing malware have the ability to overcome these limitations.

A recent study shows that behavior-based approaches to malware detection result in a higher detection rate with respect to malware complexity[39], which corresponds to the ability to directly detect malware actions on a live system via inspection of API calls. While more effective, the cost of performing such analysis involves having such an environment capable of analyzing potential malware before any infections occur.

What is clear from this research is that a combination of several techniques of malware analysis is most beneficial. Term Frequency-Inverse Document Frequency [40] metrics are used to effectively extract features of malware given access to logs of both malicious and non-malicious behavior and is robust to polymorphism [28]. Analysis of the semantics of code has been shown to detect malicious software independent from signature-based bytecode analysis [41]. The classification of particular API calls for potential malicious behavior can be used to analyze malware for particular traits [42]. We believe that a combination of these methods can aid in the trust of malware detection and threat profiling through the correlation of the resulting data.

2.3 Cyber Security Risk Management

The discovery of malware or any other cyber threat is important, but without a clear understanding of the effect of these threats there is a deficiency in the application of any of these techniques when considering risk management. However, understanding an organization's risks are not trivial as the threat landscape will differ between different organizations in different industries. Automated tools for information security risk management have been proposed [43] which an organization may use to quickly evaluate risk based upon standards such as ISO 27001. Adherence to such standards may increase an organization's resilience against risk.

Frameworks have been developed to defend against cyber security threats such as insider threat [44]. This shows that when an organization takes the time to fully assess their internal practices, behavior can be monitored for possible malicious intent. This monitoring can potentially be performed in an automated fashion through the use of threat frameworks.

2.3.1 Threat Frameworks

The Cybersecurity Framework (CSF) developed by the National Institute of Standards and Technology (NIST) [45] serves as a guide for implementing cyber security best practices. A demonstration of the effectiveness of this framework through the use of a real-world cyber attack [46] has shown that risk-based cyber security defenses can be very effective.

A methodology of assessing cyber risk in relation to the Common Vulnerability Scoring System (CVSS) [2] has been presented [47] which clearly shows how to relate the impacts of CVEs on organizational assets in terms of confidentiality, integrity, and availability. However, the dependency on CVSS is not tempered by any metric related to or provided by the organization which applies the methodology. This has the effect of promoting a "one size fits all" approach to cyber risk management – in issue we will expand upon at the end of this section.

Threat modeling and security assessment has been developed utilizing security metrics associated with discovered system vulnerabilities within software defined networks [48] according to CVSS scores. This allows for the anticipation of new threats to networked computers as a result of a potentially exploited vulnerability. Graph theory has also been used to produce a threat model which uses Markov chains in conjunction with CVSS vulnerability scores for a cloud computing environment [49].

Probabilistic threat detection for risk management in cyber-physical medical systems has been explored [50], illustrating the effectiveness of real-time threat response for such devices as a pacemaker. This research shows how predetermined threat response schemes can be highly affective for limited-purpose systems. A challenge would be to make use of probabilistic threat detection in a system with multiple and varied uses, with multiple and varied users interacting with that system.

2.3.2 Threat Scoring

Researchers have evaluated the value of threat scoring in terms of privacy risk [51], cyber threat feeds [52], CVSS metric-based analysis [53], and vulnerability assessment models based on multiple criteria utilizing CVSS [54] have been introduced, to name a few. A thorough examination of the trustworthiness of the CVSS has been presented [55] and has shown that in many instances the efficacy of CVSS-related threat models is best determined by what threat databases and criteria of threat are utilized by the end user. Indeed, the choice of resources used affects the prioritization of risks and threats. This suggests that while the “one size fits all” nature of CVSS scoring may be appropriate for vulnerabilities, the associated threats and risks must be evaluated using a separate methodology in order to be relevant for an individual organization.

This research shows the need for a robust scoring mechanism for the assessment of threats. This concept led us to consider the threat scoring system used by the Cuckoo Sandbox so as to determine what problems exist or improvements could be made to benefit the practice of information security incident response. Researchers have shown how the behavioral analysis reports from Cuckoo can be used to classify malware [31] in conjunction with labels for malware families extracted from VirusTotal [56] reports for each sample. This methodology relies on the presence of key features in the malware which correspond to a signature; however, malware authors can overcome signature-based analysis via mutation engine generating polymorphic code [57].

One question is in the determination of how to define security metrics based directly on threats, rather than vulnerabilities as with CVSS. This would require the ability for an organization to selectively rate what threat conditions they feel impact their business the most, perhaps through historical review of their cybersecurity incidents or those of their peers. Tunable security metrics have been proposed with demonstrated results [58]. In this way, the direct security concerns of an organization become the basis of determining the criticality of an incident as it relates to that organization. Utilization of threat models when designing a risk management strategy for computing systems has been shown to be effective in flexibly affecting threat values across a range of conditions [59].

2.3.3 One Size Fits All?

A one size fits all approach to cyber security auditing has been championed in some research, such as the CyberSecurity Audit Model [60]. The benefit to this approach, according to the authors, is in the use of a universal scorecard by which any organization's policies and practices can be matched to industry and government regulatory requirements. While this aspect may be effective, the authors further admit that there is no universal acceptance or standardization in terms of cyber security risk. This can mislead organizations into presuming that the only risks they need manage are related to those which are identified and controlled through compliance with industry and government regulations, which we believe is not true in the general case. Given that these regulations are concerned solely with particular types of data, there are many cyber threats which are beyond the scope of what is required for compliance. With vast differences existing in the technological and human resource structure of each organization, we believe it is necessary to consider the impact of a system of cyber security risk management which allows for bespoke threat response and prioritization.

The OCTAVE Allegro[61] methodology provides a robust means of assessing organizational cyber risks. However, it has been shown that this method's calculation of risk mitigation priorities are under-developed and can present confusion regarding which information assets are most important to mitigate first[62]. This method can be extended to include greater threat prioritization, however we believe that care must be taken to ensure that each organization's individual risk strategies are taken into account in order to truly be effective.

2.4 Challenges for Cyber Risk Management

One of the greatest challenges in cyber security is user awareness. A recent analysis of assessment approaches and maturity scales used to evaluate cyber security awareness programs [63] shows that there is a lack of published research in evaluating the effectiveness of such programs. With the threat of phishing attempts and drive-by malware infections ever present in organizations today, it would seem wise to find a way to demonstrate whether or not an organization's staff is sufficiently aware of cyber security risks.

The emerging field of study regarding computer users' wishful thinking as a response to computer threats suggests that there is much more involved in an organization's overall risk management than simple scoring mechanisms [64]. This leads us to suggest that if the scoring threat mechanism employed by an organization does not clearly map to real and credible threats, this can have a deep effect on the actions of the consumers of threat data in an organization.

Furthermore, organizations face the challenge of understanding the cyber risks they face. As businesses grow, markets change, and technology advances, organizations must consider a robust cyber risk management strategy. An excellent example of this would be the effect of many employees working from home on their personal computing devices. If a sudden change were made to allow most of an organization's staff to work remotely when previously there was limited work from home opportunities, the risks present in home networks, personal devices, and personal computer use become a risk for the organization's business computing network should these devices need to remotely connect to this network for business resources.

2.4.1 Open Source Intelligence

Open source intelligence (OSINT) provides information on cyber threats and indicators of compromise (IoCs), however it has been shown that sources of such information provide little or no processing of that data [65]. The overall effect of this lies in limitations on the usefulness of such data as there is no pre-processing to correlate the IoCs from different OSINT threat feeds. This limitation can be overcome through deduplication and IoC enrichment, creating clusters of data which relate to single threats.

It has been shown that IoCs can be extracted from cyber threat intelligence [66]. This research suggests that natural language processing can be used in conjunction with cyber threat intelligence data as an input for a risk-based cyber security framework. We believe that the benefit of this approach can be observed through a multilayered approach to cyber risk management, where multiple sources of potential threat input are used to extract threat data. A system which analyzes malware for threats against an organization is very useful and this can be enhanced with the ability to analyst OSINT data as well to provide many layers of threat analysis and risk management.

2.5 Novelty of Present Research

Many intriguing and effective solutions for cyber security threat detection and response have been described in recent research. Machine learning has been shown to classify malware with high accuracy. Natural language processing has been shown to provide the ability to make connection between API system calls and open source intelligence. Cyber security frameworks have been shown to be effective in helping organizations to understand their risks and make responsible decisions for threat management.

A problem with machine learning and natural language processing approaches lies in the expert knowledge required to utilize and modify the models over time. Not all organizations have access to such expert knowledge and will instead turn to third-party software to provide a solution. If an organization instead has the ability to provide threat and risk information to a system which automatically adjusts or retrains machine learning algorithms, the expert knowledge required of the organization will simply be in regard to its own threat and risk management, rather than deeply technical in nature.

The wealth of existing research regarding the use of deep learning techniques for malware detection provides much to build upon. Some works have investigated the malware signature analysis while others have also used sequential API calls. However, what the previous research lacks is a means of classifying malware in an online fashion. Malware changes over time due to code polymorphism as well as in response to emerging vulnerabilities in operating systems and software. The online solution presented in this work has the ability to identify evolving malware threats by learning malicious API call sequences common in malware families. Furthermore, the great amount of training time required by prior techniques makes them incapable of classifying malware through API analysis in real-time. Therefore, this work presents ELM and OS-ELM approaches that can be easily trained and do not require much computation time.

Our research provides a framework for systems designed to automate cyber security threat analysis, particularly threats involving malware. We believe that such a system would benefit organizations who lack expert technical knowledge by providing them a tool they can use in a fashion tailored for their risk profile. Previous research has shown how several different methods of malware threat discovery can be used effectively, each with their own limitations. We suggest that by combining several of these methods we can produce a system which not only discovers malware, but details the threats involved in an automated fashion so that an organization can choose the appropriate response to these threats by mapping them to an organization's risk.

This results in bespoke threat and risk response, a solution which to our knowledge has not been presented before.

Chapter 3

Cuckoo Malware Threat Scoring and Classification: Friend or Foe?

In this chapter we describe the usage of the Cuckoo Malware Sandbox, a popular open-source software tool which automates malware behavioral analysis. In particular, we focus on the malware threat scoring feature and discuss its limitations.

3.1 Introduction

While all successful malware compromises require a measure of response in order to restore faith in the proper working order of a system, not all malware attacks are created equal. For large organizations, the number of live information security incidents can be staggering. Prioritization of incidents based upon levels of severity is necessary for the quick elimination of the most severe threats and the continued monitoring and assessment of threats not yet handled. This is especially true for organizations with a relatively small information security incident response team. In situations where high volumes of security incidents may be present, an automated means of prioritization is essential to help incident analysts to quickly triage and respond in an appropriate manner. The use of automated tools, especially those made available through open-source, adds value to the work performed by the security incident response team because in many cases it allows for greater understanding of information security threats, quicker remediation times, and provides more information for after-action analysis and reporting.

The Cuckoo Sandbox [22] provides a means of automated analysis of suspicious files. Through behavioral analysis, hash comparisons, and various integrated tools it is possible to identify malware and discover what indicators of compromise one should look for in their production environment to ascertain the presence of malicious activity on a network or system. It is important to note that as an automated malware analysis system, Cuckoo provides information about the behavior of a suspected file on a system but it is not using these observations to actively classify malware. As an option, Cuckoo can be configured to enable submission of a file to VirusTotal [56] for a comparison to known malicious files and the associated malware families as determined by various anti-virus vendors. Additionally, Cuckoo can be configured to allow custom signatures and/or those from the open source community which define severity scores for particular API calls and malware family attribution [22]. These optional elements are meant to enhance the Cuckoo-generated report on a sample file so as to bring more depth to the analysis. Without these optional features Cuckoo only has the ability to execute potential malware samples in a sandbox, observe the dynamic behavior, and report the actions committed. In this thesis we are interested in the optional element of malware signatures and what value they bring to the automated malware analysis.

Another reason to consider the Cuckoo Sandbox is that it is open source software. It is familiar to the information security community [67] and free to use, which makes it more available to incident response teams who cannot easily justify the expense of similar closed-source products. While the initial setup and configuration of Cuckoo is not particularly user-friendly, the end result is a system which allows for the submission of a file and the automated return of a report describing the observed actions of a suspected malware sample in a sandbox environment. The benefit of Cuckoo is in this automation – no human interaction is required for the behavioral analysis. A report is generated describing the actions which occurred when the suspicious file was opened or executed, and all an analyst needs to do is investigate the report. A detailed list of system operations, file manipulations, attempted communication with external systems, and even screenshots of the activity are generated. All of these artifacts are essential for understanding the nature of the malicious file analyzed.

One element of the Cuckoo report stands out as an element which immediately captures the eye of an analyst, and for good reason. The “Score” section appears at the top of the report and represents the threat severity of a malicious file as a number out of ten. The color of the section containing the score changes with severity, further suggesting the impact of a file as it goes from benign light green with a score of 0/10 to an angry red with a score of 10/10. This score does come with a notice from the developers: “Please notice: The scoring system is currently still in development and should be considered an alpha feature.”

While using Cuckoo as an analysis tool in practice, we discovered odd behavior in the reported score for many malware samples. Several executable files were identified as malware with a score higher than the upper threshold of 10. This was confusing and caused us to wonder if there was an error in the scoring mechanism. Instead we discovered that there was no error in the configuration of our system or a bug in the Cuckoo software. Instead, we found that the methodology used by Cuckoo to generate this threat score results in an arbitrary value that is of little help to illustrate the threat severity of malicious code to an incident responder. The arbitrary nature of this score is not immediately apparent to the end user. The otherwise excellent value of the Cuckoo Sandbox in automated behavioral analysis might lead an incident responder to place a similar value in this score, perhaps so far as to reduce the priority of remediation for an incident involving a malware with a score of 2/10 in comparison to a similar incident involving a malware with score of 9/10 or 15/10. This could also lead an incident responder to prioritize an incident involving less actual risk, given the arbitrary nature of the threat score.

3.2 Malware Behavior Analysis

In order to evaluate the current scoring mechanism Cuckoo uses to classify severity, we designed an environment to allow for the installation of Cuckoo and the analysis of known malware samples in a virtual machine sandbox per the installation instructions provided by Cuckoo [22]. Our goal was to create an environment which would perhaps be most typical for small to medium-sized security incident response teams and focus on the evaluation of potentially malicious software affecting Windows operating systems. This included the usage of a single Linux host machine to run the Cuckoo application and house Windows 7 virtual machine guest environments. Cuckoo allows for the configuration of Linux virtual machine guest environments as well; however, for this work we were most interested in the analysis of Windows-based threats.

3.2.1 Setup

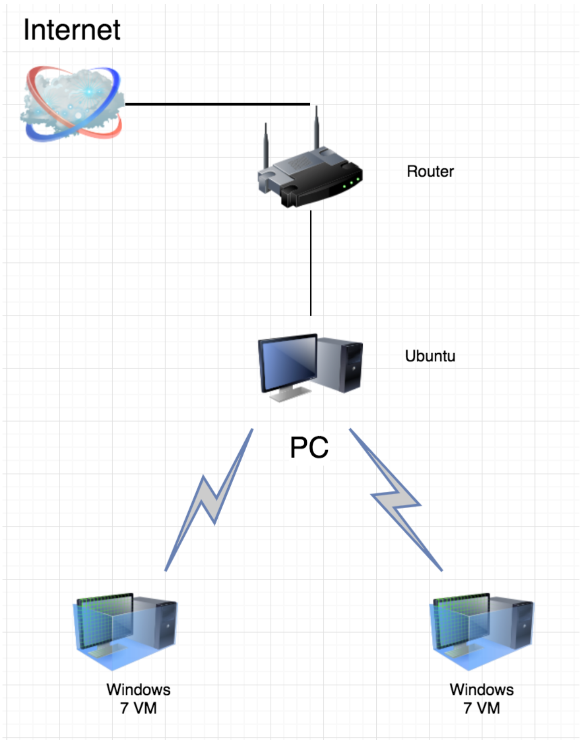


FIGURE 3.1: Setup for Cuckoo Sandbox and the VM Environment.

Cuckoo Sandbox was installed on a dedicated Ubuntu Linux host with access to the public internet, as shown in figure 3.1. Cuckoo was configured per the installation guide found on the Cuckoo website [22], including two 64-bit Windows 7 virtual machines installed on the Ubuntu host. Cuckoo recommends the usage of 64-bit Windows 7 over Windows XP for better results. Cuckoo supports many virtualization software solutions but does assume the usage of VirtualBox by default, so for ease of setup we chose this platform. VirtualBox is a free system virtualization product developed by Oracle and it easily integrates with Cuckoo for administration of the virtual machines.

The Windows 7 virtual machines were configured with essentially none of the built-in security measures in order to make them as vulnerable to malware as possible, including the disabling of User Access Control, Windows Firewall, and automatic updates. Software known to be the target of attack for malware was also installed on the virtual machines, including Adobe PDF reader, Java, and Microsoft Office. This ensures that when a malicious software sample is executed in the environment, the full breadth of the malware's behavior might be observed as it may attempt to access these applications as part of the process to compromise the system. Python has been installed to facilitate the communication between the Windows operating system on the virtual machine and Cuckoo on the host machine. Afterward, the Windows virtual machine resets to the base, non-compromised state via an automated Cuckoo command to VirtualBox.

Furthermore, on the host Ubuntu system we ran the "cuckoo community" command to load the signatures provided by contributions from the Cuckoo user community. These signatures are curated by the Cuckoo development team and provide the definitions of malware severity and family attribution described later in this Section.

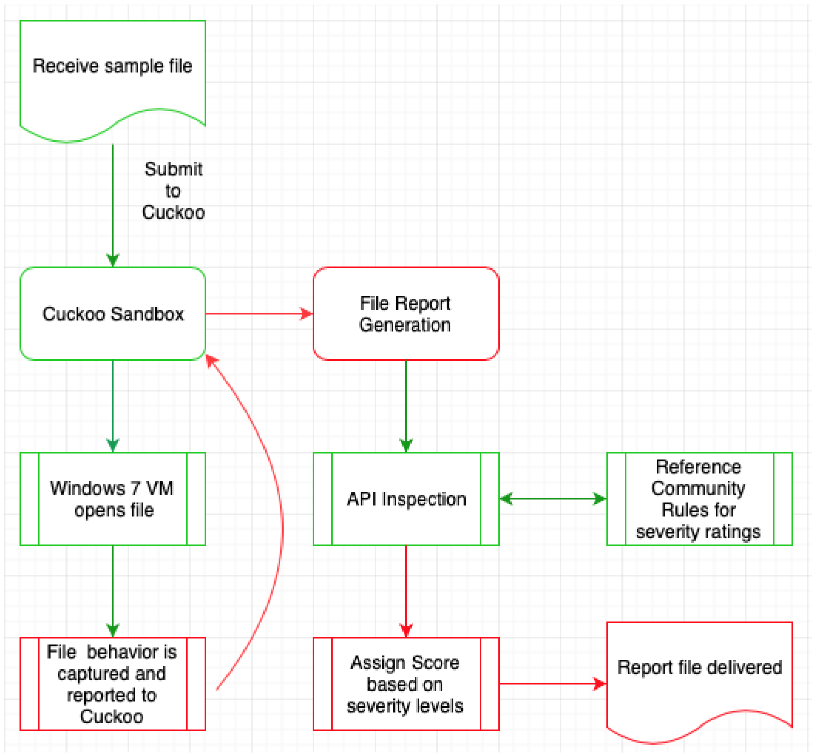

Receive sample file
Submit to Cuckoo
Cuckoo Sandbox
File Report Generation
Windows 7 VM opens file
API Inspection
Reference Community Rules for severity ratings
File behavior is captured and reported to Cuckoo
Assign Score based on severity levels
Report file delivered

FIGURE 3.2: Flow of the Cuckoo Sandbox malware analysis and report generation.

3.2.2 Malware Dataset

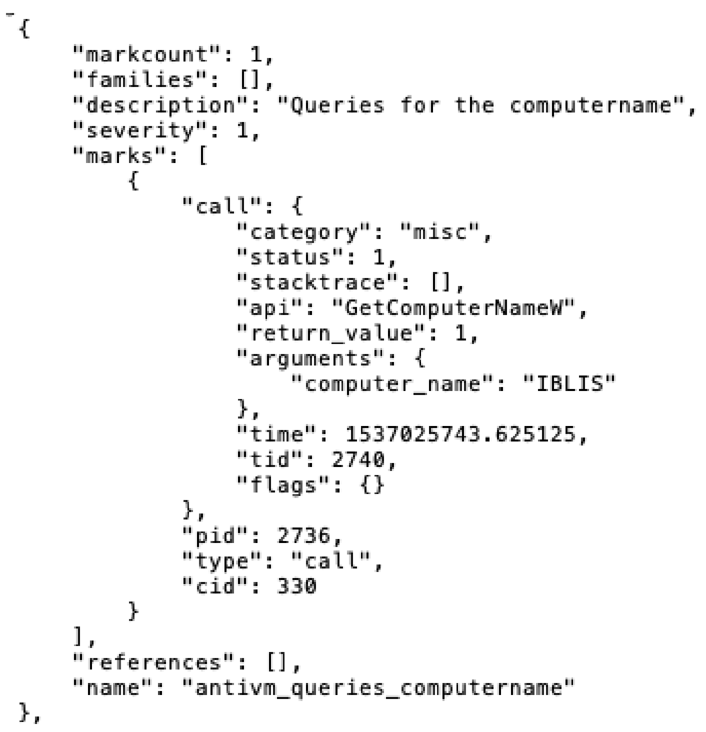
Known malware samples were acquired from Malpedia [68], a curated online resource of malicious software containing multiple versions of malware samples seen over time. This allows for the observation of evolving behaviors as the methods of exploiting system and application vulnerabilities changes with new generations of malware. Malpedia samples often include references to third party analysis of the malware as well as identified malware family and threat actor affiliation. This information is quite valuable for those desiring to create custom signatures in Cuckoo for malware family attribution. However in this work we were mainly concerned with the native functionality of Cuckoo scoring using the community-provided signature set.

3.2.3 Methodology

Figure 3.2 describes the process of malware analysis in the Cuckoo Sandbox. When a malware sample is submitted to Cuckoo, it will designate a virtual machine for use in analyzing the software. The virtual machine will resume from a snapshot from which it was in a known good, non-compromised state and Cuckoo passes the malware sample to the virtual machine for execution and analysis. Once the analysis has been performed, Cuckoo generates a report of the observed activity, including but not limited to changes to the registry, newly spawned processes, file creation and access, virtual memory access, HTTP communication to an external IP, and much more. These behavioral events are captured as a number of Windows API calls.

These API calls are each designated a severity score, which is determined by a repository of rules defined by the Cuckoo user community. According to the Cuckoo documentation [22], the range of severity scores is from 1-3, though our observation shows scores of 5 as well. While often a description will be included in the signature asserting the malicious nature of how a particular API call is being manipulated by malware, it is not immediately clear how the actual severity value is determined. All of the API calls flagged in these signatures perform common actions in a Windows system. This makes ranking a particular API call as more suspicious than another quite difficult. The community signatures clearly are attempting to note API calls which are used to have a potentially damaging effect, but the ranking from 1-3 or higher appears to be at the discretion of the author of the signature and subject to review by the Cuckoo developers.

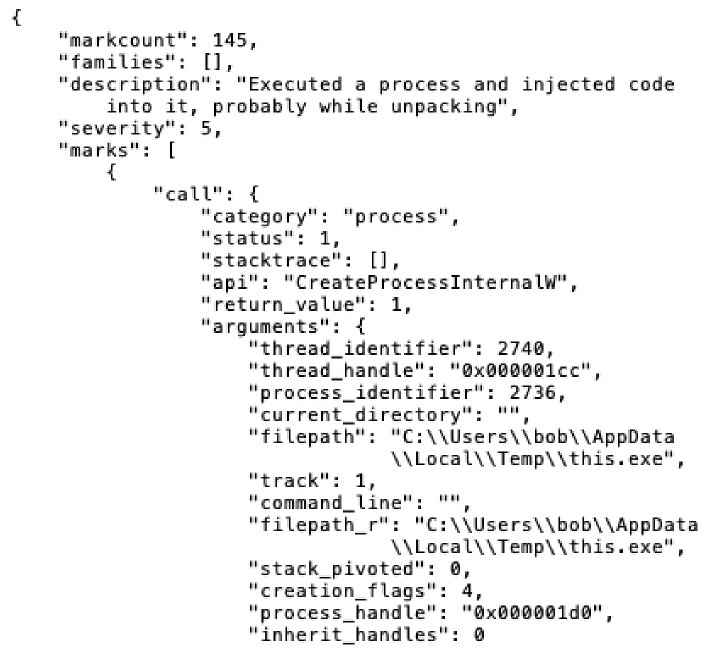
Once all of the behavior witnessed by Cuckoo has been checked against these signatures and severity scores have been tallied, a final report is generated and available for inspection by an analyst via local web page or command line delivery.



```
{
    "markcount": 1,
    "families": [],
    "description": "Queries for the computername",
    "severity": 1,
    "marks": [
        {
            "call": {
                "category": "misc",
                "status": 1,
                "stacktrace": [],
                "api": "GetComputerNameW",
                "return_value": 1,
                "arguments": {
                    "computer_name": "IBLIS"
                },
                "time": 1537025743.625125,
                "tid": 2740,
                "flags": {}
            },
            "pid": 2736,
            "type": "call",
            "cid": 330
        }
    ],
    "references": [],
    "name": "antivm_queries_computername"
},
```

FIGURE 3.3: Sample of "GetComputerNameW" API call with a severity score of 1.

Figure 3.3 presents an example of an API call with a severity score of 1. The activity shown here is the usage of the GetComputerNameW Windows function which retrieves the NetBIOS name of the local computer. The signature matched is identified as "antivm_queries_computername" and the function of this call is indeed to query for the name of the computer. In this example, our Windows 7 VM chosen by Cuckoo to analyze the malware sample was named “IBLIS.” For the full set of sample malware we analyzed, we found that this particular API call was committed a total of 38,867 times.



```
{
    "markcount": 145,
    "families": [],
    "description": "Executed a process and injected code
        into it, probably while unpacking",
    "severity": 5,
    "marks": [
        {
            "call": {
                "category": "process",
                "status": 1,
                "stacktrace": [],
                "api": "CreateProcessInternalW",
                "return_value": 1,
                "arguments": {
                    "thread_identifier": 2740,
                    "thread_handle": "0x000001cc",
                    "process_identifier": 2736,
                    "current_directory": "",
                    "filepath": "C:\\Users\\bob\\AppData
                            \\Local\\Temp\\this.exe",
                    "track": 1,
                    "command_line": "",
                    "filepath_r": "C:\\Users\\bob\\AppData
                            \\Local\\Temp\\this.exe",
                    "stack_pivoted": 0,
                    "creation_flags": 4,
                    "process_handle": "0x000001d0",
                    "inherit_handles": 0
```

Figure 3.4: Sample of ”CreateProcessInternalW” API call with a severity score of 5.

Figure 3.4 presents an example of a CreateProcessInternalW API call which has been classified by the community rules with a severity score of 5. This particular call demonstrates the malware executing a process and injecting code into it. In this example, a process identified as 2736 is being loaded with the code within the file “this.exe” from within the logged in user’s temporary files within an AppData subdirectory. It is interesting to note that the description of the behavior matched in the signature involves the possibility of code unpacking, which refers to a compressed executable file which must “unpack” in memory in order to execute. This procedure of unpacking is common in malware samples so it is likely that this is why the severity score for this signature was rated as a five. However, there is no clear indication that this is the case, which further leads to confusion regarding the assignment of this value.

We discovered that this particular API was called 32,880 times across our sample set. The severity score of each identified signature-matched activity is evaluated and a final “Score” is produced in the report. Not all of the signatures single out specific API calls, as it has been seen that certain signatures refer to specific known malicious sites, known malicious malware file names, etc.

3.2.4 Discussion

Cuckoo reports were generated for 7,401 known malware samples retrieved from Malpedia. Analysis of these reports show that there were 138,523,300 API calls in total with just 264 unique API calls in all. Given that the Cuckoo analysis assigns severity scores to particular API calls as a result of a signature match, we found it interesting to compare the frequency of API calls to the overall report threat score reported by Cuckoo. The severity ratings for these API calls are determined by Cuckoo community signatures, but not all of the APIs discovered in a Cuckoo analysis match these signatures. Thus we find that a high number of API calls will not necessarily translate to a high report threat score.

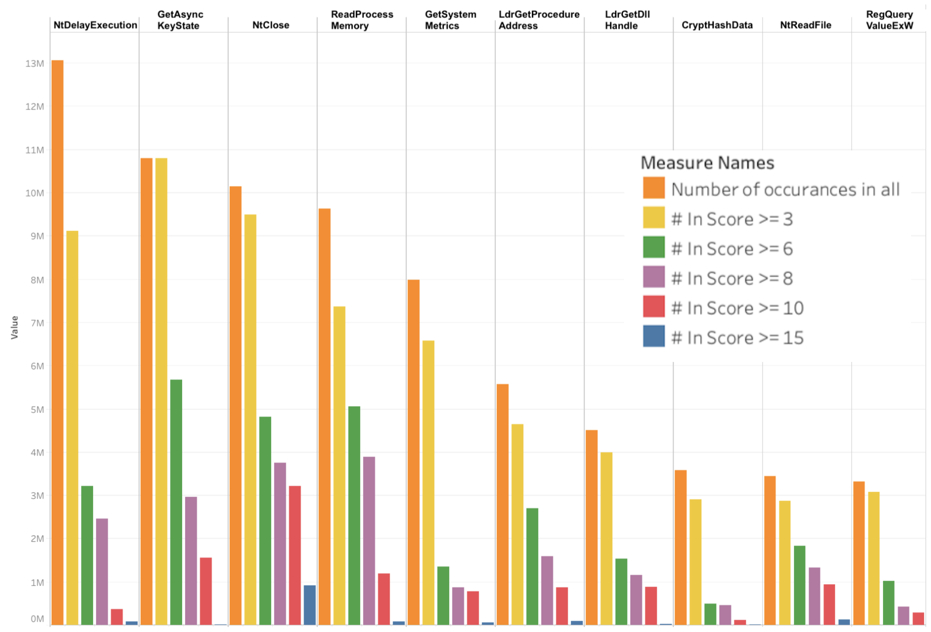


Figure 3.5: Sample frequency of API calls by number of samples in Cuckoo report score range.

As seen in Figure 3.6, our sample file named 891 had a reported 1,326,166 individual API calls (174 unique) and was assigned a threat score of 16.8/10 by Cuckoo. In contrast, our most threatening file with a score of 23.6/10 committed only 34,480 API calls (190 unique).

What we discovered is that the majority of these individual function calls do not result in a severity score because there is no associated signature to assign the value. In fact, a single API function call may result in different severity rating assignments based upon different signature matched.

For example, one malware sample was observed calling the "NtAllocateVirtualMemory" function with this behavior of possibly allowing code injection for another process and was assigned a severity score of 3. In the same malware sample this API function call was made again with the behavior matching a rule describing the possibility of the injection of code into an executed process while unpacking, resulting in a severity classification of 5. This creates a challenge when attempting to determine the relative threat of a particular API call observed in a malware sample. We believe a great deal of confusion regarding consistent threat classification exists given the disparity between severity scores for the same API calls amongst the signatures.

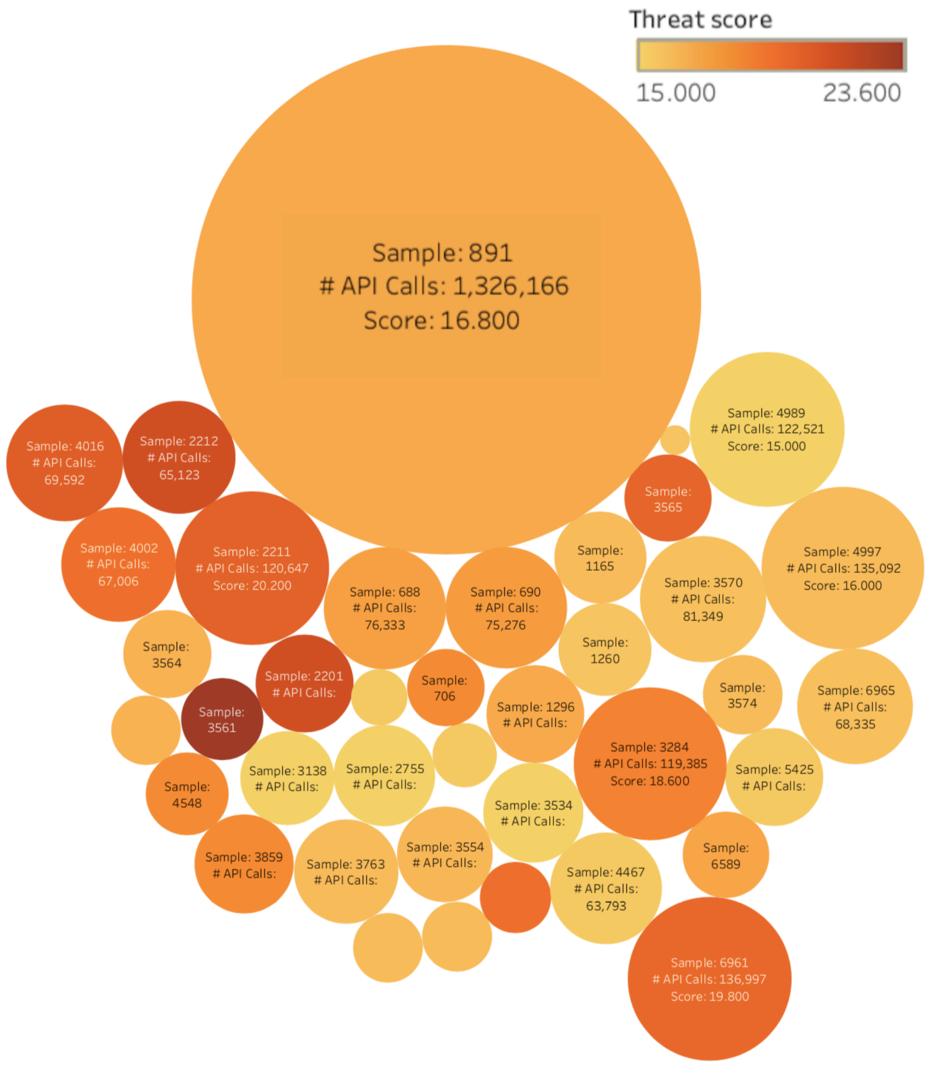


Figure 3.6: Distribution of malware samples with score of 15 and greater, sized by the number of API calls.

Table 3.1 describes this relationship for the top ten occurring API calls.

Table 3.1: Top Ten Occurring API Function Calls

Function	Type	Occurances
NtDelayExecution	Windows Kernel	13,052,984
GetAsyncKeyState	Windows Control	10,786,787
NtClose	File System	10,142,832
ReadProcessMemory	Process Control	9,626,589
GetSystemMetrics	Windows Conrol	7,974,563
LdrGetProcedureAddress	Windows Kernel	5,563,886
LdrGetDllHandle	Windows Kernel	4,497,628
CryptHashData	Security & Identity	3,568,810
NtReadFile	File System	3,425,244
RegQueryValueExW	Windows Registry	3,303,343

Figure 3.5 details the relationship between these highest occurring API function calls and the final threat score Cuckoo assigned to malware samples containing these calls. It is interesting to note the changes in frequencies of these API calls between the different threat values. For instance, "GetAsyncKeyState" is called a total of 10,786,459 times in the malware samples Cuckoo rated a threat score greater than or equal to three and just 1,543,317 for malware samples greater than or equal to ten.

Of all of our tested malware samples, the average score was 2.6 with the majority of samples with a score in the 0 – 2.12 range. Of particular interest is that we found that 257 samples were rated above the suggested maximum "10" rating. These ranged from 10.2 – 23.6. This finding is of interest because nearly 3.5% of the samples analyzed were classified a threat rating above the maximum value. Why was this the case and what does this mean about the severity of these samples in comparison to those with a threat rating below 10? The answer to our question is that the arbitrary nature of the threat score denies the opportunity for the 0 – 10 rating to provide an evaluation of the threat of a given malware sample in relation to others. Essentially, what we can say about any suspected malware sample with a score of 0 is that no behaviors matched a signature rule. A score of 0.2, or 1/5, implies that exactly one signature was matched for an observed behavior. Since any higher score is similarly only representative of the number of potentially malicious actions and not truly representative of the impact of the risks associated with those actions, we feel that this is an insufficient metric for assigning priority of malware remediation tasks for an incident response team.

3.3 Evaluation

3.3.1 Results

Given that in this sample set we observed malware threat severity scores from 0 – 23.6, we decided to understand the methodology behind the value of the threat score. This required the analysis of the reports for each malware sample in our set which was evaluated by Cuckoo Sandbox. These reports are easily parsable as they are in JSON format. We found that the bulk of the Cuckoo report involves the observed behavior of the malware in terms of Windows API calls. Many of these API calls were designated a “severity” score within this report and it is here that we see what Cuckoo uses to determine the final report threat score value. When creating the report for a sample, captured behaviors are compared to any signature rules enabled as part of the Cuckoo configuration. The final threat score assigned by Cuckoo to the analyzed malware is the result of adding the scores of all the signatures which match an observed behavior of the malware and then dividing the sum by 5. Therefore, if S_o^n is the threat rating of a signature which matches the n-th observed behavior of the malware then the malware's final threat score S_f is given by:

$$S_f = \frac{\sum_{n=1}^{k} S_o^n}{5} \tag{3.1}$$

where k is number of observed malware behaviors that matched a cuckoo signature. For example, in the highest rated malware sample in our testing set Cuckoo observed six behaviors which matched signatures with a severity rating of 1, sixteen behaviors which matched signatures with a severity rating of 2, twenty-five behaviors which matched signatures with a score of 3, and one behavior which matched a signature with a severity rating of 5. The sum of these severity ratings is 118, which when divided by five results in the final threat score of 23.6 assigned by Cuckoo. We observed this formula again when analyzing a malware sample with 1,326,166 observed API function calls, with only 36 of these API calls resulting in a signature match – eight with a severity of 1, ten with a severity of 2, seventeen with a severity of 3, and finally one behavior matching a signature with a severity score of 5. The sum of these severity ratings is 84, which when divided by five results in the final threat score of 16.8 delivered in the Cuckoo report. The number five is the consistent denominator across all malware samples regardless of the number of APIs called or signatures matched. The developers of the Cuckoo Sandbox acknowledge the use of this number five[69] and state that this means of generating a final threat score is in need of improvement[70].

There is no obvious benefit in the use of the number five to divide the total severity score to achieve the final threat value for a malware sample. For all 7,401 malware samples, a sum was tallied for every severity score in a sample report and divided by five to achieve the same final threat score reported by Cuckoo. The consistent use of the number five across all analyzed malware sample suggests that this number is not related to any behavior or attribute of the analyzed malware samples but is instead an arbitrarily chosen value, similar to the severity values assigned to the Cuckoo community signatures. This adds to the confusion regarding the true value of the threat reported for a malware sample by Cuckoo.

What we found is that Cuckoo is designed to produce a final threat score which is the sum of the severity ratings for each observed API call, divided by five. What is not immediately obvious is what this means for the relationship between different malware samples analyzed in Cuckoo. The denominator value of five used in the calculation of the final threat score is not associated with the number of API function calls or other behaviors observed by Cuckoo in its behavioral analysis, nor is it linked to the number of matched signatures. Therefore, we cannot take this final threat score as an accurate measure to compare malware samples for the relative priority by which they should be addressed to mitigate the risk which they represent.

3.3.2 Discussion and Recommendations

The arbitrary nature of the Cuckoo scoring methodology casts confusion upon the threat level of a given malware sample. For the incident responder, it might seem appropriate to immediately act to contain an incident involving a malware with a score of 18.8 instead of an incident involving a malware with a score of 6.4. However, if it is determined that the malware with the larger score is simply repeatedly attempting to reach out to a dead host on the internet (and of course failing) while the malware with the lower score is actively injecting malicious code into legitimate Windows services, this casts confusion upon the meaning of the severity score. Should such a score be evaluated in conjunction with the behavior to determine actual severity? One could make a case for ignoring the scoring mechanism altogether and focus on the relevant indicators of compromise.

Yet the purpose of a threat score is to provide a quick, immediate index value to support effective triage. Even though the current methodology used in the Cuckoo Sandbox is arbitrary, one could argue that it was included to provide a general guide to help support a quick assessment. What is necessary is a more reliable metric. Currently the Cuckoo report threat score is dependent upon community-provided signatures. The severity score assigned to these signatures affects the final score Cuckoo evaluates, again via an arbitrary method. Of note is the fact that if the community signatures are not loaded into Cuckoo as described in Section III and no custom signatures are created by the user, each file analyzed by Cuckoo will have a threat score of zero due to not having any API severity values to evaluate.

This issue is not limited to the problem of scoring. Malware family classification is also dependent on these community signatures. Thus when Cuckoo analyzes a malware sample and reports that it belongs to a particular malware family, this is determined by the logic provided in a signature which checks for the presence of a particular mutex or specific filename in a particular Windows folder, for example. This is not a particularly robust method as malware behavior will change over time with new generations and variants.

We believe that in order to take the confusion out of the malware threat scoring, a more thorough analysis of malicious API calls is necessary. Statistical analysis of these function calls shows that a large amount of activity witnessed by Cuckoo remains unevaluated. Instead of relying on signatures, the breadth of the API calls can be evaluated in conjunction with the context of the behavior. For example, an API call made to reach an external host on the internet which is known to be a recent command and control server used by a large threat actor should carry a heavier severity score than a similar API call reaching out to Google. This requires both statistical analysis of each malware sample as well as reliable, current threat intelligence. In this manner, threat scoring and malware family attribution will be dynamic, robust, and provide greater confidence in threat prioritization for the incident response team.

Chapter 4

Insights Into Malware Detection via Behavioral Frequency Analysis Using Machine Learning

In this chapter we describe how several machine learning techniques can be used to discriminate malicious from benign software by virtue of analysis of operating system Application Programming Interface (API) calls made by software on a system within which it was executed. These API calls are observed through Cuckoo as described in Chapter 3 and both linear and non-linear machine learning models are used to make these distinctions.

4.1 Introduction

In our research we found a lack of analysis regarding the meaning of why several different machine learning approaches can identify malware with high accuracy on specific datasets. It is not always clear how machine learning models for detecting malware address the issues of false positive and false negative classification. We show that malware and benign software contain many of the same Windows API function calls which define their behavior on a host computer. Analysis of only the presence of these function calls is therefore not sufficient to divide malware from benign software. In this work we will show that analysis of the frequencies of these API calls provides a powerful means of understanding the difference between malicious and benign activity. Furthermore, our interest is in observing what machine learning algorithms can teach us about malware as well as our understanding of how to identify malware. Our research leads us to discover the relationship between the system calls used as features in the models. This work presents a first step on the path to revise the way we think about the behavior of malware and what that could mean for the next generation of malware analysis systems.

4.2 Malware Behavior Analysis

In order to programmatically observe the behavior of malware in an isolated environment, we designed an environment to allow for the installation of Cuckoo and the analysis of known malware samples in a virtual machine sandbox per the installation instructions provided by Cuckoo[22]. Our goal was to focus on the evaluation of potentially malicious software affecting Windows operating systems.

4.2.1 Setup and Malware Dataset

Cuckoo was configured per the installation guide found on the Cuckoo website[22], including two 64-bit Windows 7 virtual machines installed on the Ubuntu host. Cuckoo supports many virtualization software solutions but does assume the usage of VirtualBox by default, so for ease of setup we chose this platform. VirtualBox is a free system virtualization product developed by Oracle and it easily integrates with Cuckoo for administration of the virtual machines.

Known malware samples were acquired from Malpedia[68], a curated online resource of malicious software containing multiple versions of malware samples seen over time. This allows for the observation of evolving behaviors as the methods of exploiting system and application vulnerabilities changes with new generations of malware. Malpedia samples often include references to third party analysis of the malware as well as identified malware family and threat actor affiliation. This information is quite valuable for those desiring to create custom signatures in Cuckoo for malware family attribution.

4.2.2 Methodology

Once the analysis has been performed, Cuckoo generates a report of the observed activity, including but not limited to changes to the registry, newly spawned processes, file creation and access, virtual memory access, HTTP communication to an external IP, and much more. These behavioral events are captured as a number of Windows API calls and can be referenced programmatically through created JSON files.

One example of a Windows API call is the GetComputerNameW function. The activity shown here is the usage of the GetComputerNameW Windows function which retrieves the NetBIOS name of the local computer. The signature matched is identified as "antivm_queries_computername" and the function of this call is indeed to query for the name of the computer. For the full set of malware samples we analyzed, we found that this particular API call was committed a total of 38,867 times.

4.3 Machine Learning Approach

Our dataset consists of the Windows application programming interface (API) function calls observed by our Cuckoo malware behavioral analysis environment as described in Section III. The API calls recorded in our dataset represent the activities performed by 7,401 malware samples. Windows API functions are called by applications in order to operate in a Windows environment [71]. Analysis of API calls made by an application in a Windows environment therefore presents a concrete record of all behaviors performed by that application.

In our behavioral analysis we identified a number of Windows API calls and their frequencies which correspond to the actions performed by malware on a system, such as registry changes, code injection into running processes, file modification, etc. It is important to note that it is not trivial to analyze API calls for malicious behavior. In addition to known malware samples, we also performed behavioral analysis against a set of thirty known benign software samples.

These benign software samples consist of a mixture of application installers, Java applications, Microsoft Word documents, and similar executable files. Analysis of malicious and benign software samples in Cuckoo resulted in the observation of over 138 million API calls, with 264 unique referenced functions. Of these 264 API calls we found that 84 unique API functions were found in malware but not in benign software.

Unfortunately, these 84 APIs were called in only a small fraction of the malware samples and therefore are not useful as a “smoking gun” for determining if a given application is malicious if its behavior includes these particular function calls. Therefore, we concluded that the best usage for applying a machine learning model in malware analysis would be to examine the relative frequencies of these 264 APIs across malicious and benign software. It is important to note that the frequency of the API calls largely vary. For example, we found a greater number of the "GetAsyncKeyState" Windows API call in comparison to the "GetCursorPos" call. This is likely due to a greater interest in software to react to a computer user's mouse button usage[72] than in the screen coordinates of the mouse cursor[73]. This suggests that while particular Windows API functions may be selected as features in a machine learning model to identify malware, there is reason to analyze what such a decision really means for understanding the behavior of the identified malicious software. Our approach was to apply several machine learning algorithms to three different sized malware sample sets and compare them. Our goal was to identify the algorithmic approach which should most reliably classify malicious software from benign software through analysis of the frequency of Windows API calls.

4.3.1 Methodology

Eight machine learning algorithms were evaluated on the set of values describing the Windows APIs called for each malicious and benign sample:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- K-Nearest Neighbors (KNN)
- Classification and Regression Trees (CART)
- Gaussian Naive Bayes (NB)

- Support Vector Machines (SVM)
- Decision Tree
- Random Forest

Our desire was to use a mixture of linear (LR and LDA) and non-linear (KNN, CART, NB, and SVM) algorithms to determine which would be good for our dataset. The applicability of classification trees to relationship models led us to further break CART down into Decision Tree and Random Forest classifiers using the Scikit-Learn Python library for visualization as shown in[74]. Our implementation included Python and the SciPy platform using a random number seed which was reset before each run to ensure that the results were directly comparable as shown in[75].

Through the use of a Python program we were able to compare each of these algorithms as they attempted to classify a given software sample as malicious or benign through the evaluation of the relationships between the API frequencies in known malicious and benign software samples. Once such a training model was created for each algorithm, these models were used to classify a testing subset of unevaluated software. The accuracy of the training models against the test subset for each algorithm was then compared to determine the best performing algorithm. In addition to

Table 4.1: Three Datasets

Dataset	Malware Count	Benign Count
Large Sample	7,400	30
Medium Sample	853	30
Small Sample	30	30

evaluating different machine learning algorithms, the decision was made to evaluate each upon different sample sizes of malicious and benign software. The Large Sample dataset contains the API frequencies of all 7,400 malware samples from our known malware dataset along with the API frequencies of thirty known benign software samples. The Medium Sample dataset consists of 853 randomly selected known malware samples along with our thirty known benign software samples. The Small Sample dataset consists of an equal number of known malicious and known benign samples, again with a randomly selected subset of the original 7,400 malware samples. Table 4.1 summarizes the three datasets used to test the accuracy of the algorithms against different ratios of known malicious to known benign software.

We utilized a 70/30 split for training and validation for each dataset, which trains each machine learning model on 70% of the dataset and then tests 30% to determine accuracy. 10-fold cross validation was used to estimate the accuracy of the machine learning algorithms.

Given its high relative accuracy across our datasets, we chose to further evaluate our models generated with KNN to break down their confusion matrices. A confusion matrix considers the total number of elements in the validation dataset and then further classifies these elements into the following categories: true positive, false positive, true negative, and false positive. Since we are attempting to classify malware, a true positive result would reflect that an actual malware was identified as malicious.

Likewise, a false positive would involve the classification of benign software as malicious. Therefore, our most successful models would generate the highest number of true positives and true negatives.

Additionally, we also observed high accuracy with CART. This intrigued us due to the possibility of relationship modeling using decision trees and lead us to include a deeper dive into the accuracy of different decision tree models. Our Decision Tree implementation utilizes two methods for determining a root node with the goal of comparing these two methods for different accuracies. Equation 1 describes the Gini Index, which is a metric used to determine how often a randomly chosen feature would be incorrectly identified. In our case, each API frequency is evaluated as a feature and the one with the lowest Gini Index is identified as an appropriate root node for a tree.

$$Gini(E) = 1 - \sum_{j}^{c} P_j^2 \tag{4.1}$$

Equation 2 describes our second method for determining a root node for a tree. Here entropy is used to determine the impurity of a feature as a measure of information gain. In this way the feature with highest calculated entropy will be selected as the root node of a tree.

$$H(X) = -\sum_{j}^{c} p(x_i) log_2 p(x_i) \tag{4.2}$$

4.3.2 Results

4.3.2.1 Large Sample

Table II shows most of our algorithms are extremely close to purporting 100% accuracy. This is likely due to the much larger number of malware samples included in this dataset skewing the learning algorithm's perception to understand mostly malicious behavior. This leads us to conclude that four models for the Large Sample are overfit, with the notable exception of NB which appears to under-perform for each of our datasets. We also observed that KNN gave us 9 false positives and 2,220 true positives according to the model. Figure 4.1 illustrates the Decision Tree model for

Table 4.2: Comparison of Algorithms for Large Sample

Linear/Non-linear	Algorithm	Mean	Standard Deviation
Linear	LR	0.995193	0.002470
Linear	LDA	0.981541	0.010784
Non-linear	KNN	0.995963	0.002499
Non-linear	CART	0.995001	0.002878
Non-linear	NB	0.241691	0.023089
Non-linear	SVM	0.995963	0.002499

the Large Sample as a series of branching nodes. Each node describes a value for a particular Windows API, followed by the Gini Index value, the number of software samples (malicious or benign) which made a call to this particular Windows API, a tuple which delineates the number of benign software from the number of malicious software samples, and finally a classification of Malware or Benign as predicted by the learning model.

The Decision Tree model for the Large Sample reported 99.59% accuracy with gini index with one true negative, three false negatives, eight false positives, and 2,217 true positives. We also observed zero true negatives, zero false negatives, nine false positives, and 2,220 true positives with 99.6% accuracy using entropy. Utilizing Random Forest for the Large Sample we observed two true negatives, zero false negatives, seven false positives, and 2,220 true positives with 99.69% accuracy.

The decision trees created show a pronounced bias toward identifying malware as opposed to identifying benign software samples. This is likely due to the much higher amount of malware samples in the Large Sample. This apparent overfit is what led us to continue evaluating these algorithms with a smaller dataset.

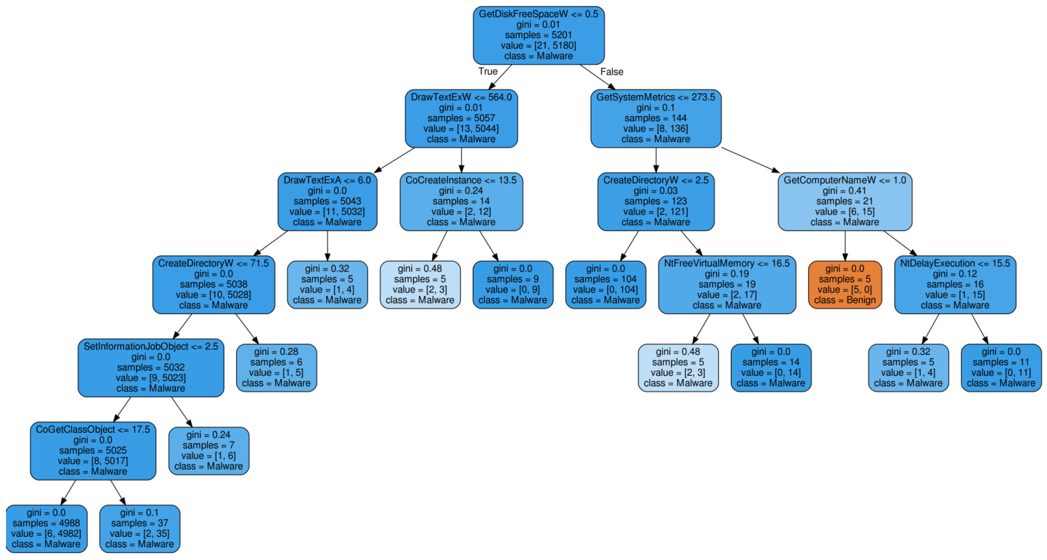


Figure 4.1: Decision Tree for Large Sample

4.3.2.2 Medium Sample

In the Medium Sample we observe results which appear to be much less overfit than what was seen from the Large sample. Table III shows a much different algorithm comparison than what was seen for the Large Sample. Of interest is that NB performed similarly to the Large Sample, with a much lower accuracy than the other algorithms. As with the Large Sample we see that KNN and SVM perform with the highest accuracy. The confusion matrix for KNN shows us that this algorithm gave us one true negative, three false negatives, ten false positives and 251 true positives according to the model. Our Decision Tree model decreased accuracy by approximately 8% using Gini, with four true negatives, sixteen false negatives, seven false positives, and 238 true positives. This is in contrast to our model using entropy, which decreased by approximately 2% with nine true negatives, four false negatives, two false positives, and 250 true positives. This suggests that the use of entropy to determine a root node works much better than the Gini approach for the API frequency data being categorized. Our Random Forest model decreased accuracy by approximately 4% with one true negative, one false negative, ten false positives, and

Table 4.3: Comparison of Algorithms for Medium Sample

Linear/Non-linear	Algorithm	Mean	Standard Deviation
Linear	LR	0.954654	0.027075
Linear	LDA	0.881914	0.058108
Non-linear	KNN	0.964410	0.020299
Non-linear	CART	0.959572	0.025247
Non-linear	NB	0.737916	0.060903
Non-linear	SVM	0.966023	0.019854

253 true positives. The lack of true negatives in this result suggest that either a small amount of benign software was provisioned into the test dataset or this model continues to suffer from overfit.

Figure 4.2 illustrates a decision tree generated by our model for the Medium Sample. Here we see quite a difference from the tree for the Large Sample, as there is much less bias toward identifying malware versus benign software. This tree appears to not suffer from as much overfit as we saw in the Large Sample. As a whole, the machine learning algorithms created models which seem to be more accurate for the Medium Sample than for the Large Sample.

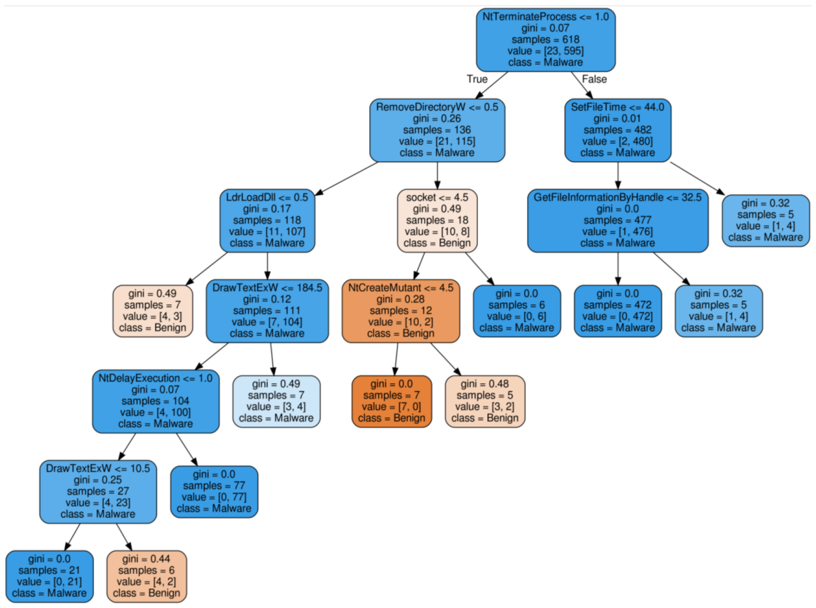


FIGURE 4.2: Decision Tree for Medium Sample

4.3.2.3 Small Sample

For the Small Sample we observe that our models are underfit. We have a proportionately high ratio of true negatives to true positives reported, along with a relatively higher number of false negatives. The overall accuracy of our models has reduced significantly as a result. Table IV shows a greater amount of outliers as opposed to our previous sample datasets, reducing the overall accuracy. Of interest is that NB has increased in accuracy in comparison to its performance in the larger datasets while others, especially SVM have greatly reduced accuracy. As noted earlier, KNN and SVM had previously performed with comparable accuracy. Here we can see that KNN performed with 36% greater accuracy than SVM. Furthermore, KNN gave us eight true negatives, six false negatives, zero false positives and four true positives according to the model. Interestingly, as seen with the Smaller Sample we note that

TABLE 4.4: Comparison of Algorithms for Small Sample

Linear/Non-linear	Algorithm	Mean	Standard Deviation
Linear	LR	0.695	0.180901
Linear	LDA	0.67	208806
Non-linear	KNN	0.715	0.264622
Non-linear	CART	0.81	0.185472
Non-linear	NB	0.79	0.115758
Non-linear	SVM	0.355	0.180901

here our decision trees created with entropy have greater accuracy than those using the Gini Index. Our decision tree made with Gini had an accuracy of approximately 83.33% with a confusion matrix showing seven true negatives, two false negatives, one false positive, and eight true positives. Comparatively, our model using entropy resulted in an accuracy of approximately 94.44% along with eight true negatives, one false negative, zero false positives, and nine true positives. This suggests that as the sample datasets become smaller and the ratio between malware and benign software samples becomes more equal, entropy is a much better deciding measure for a root node. Furthermore, our Random Forest model suffers from reduced accuracy with this Small Sample dataset. Eight true negatives and eight true positives are offset by two false negatives and zero false positives. There is a progressive increase in false negatives as our sample size and disparity between malicious and non-malicious software samples has decreased. Our Decision Tree for the Small Sample illustrates a bias toward classifying software as benign. This indicates the increase in false negatives as our models are now clearly underfit.

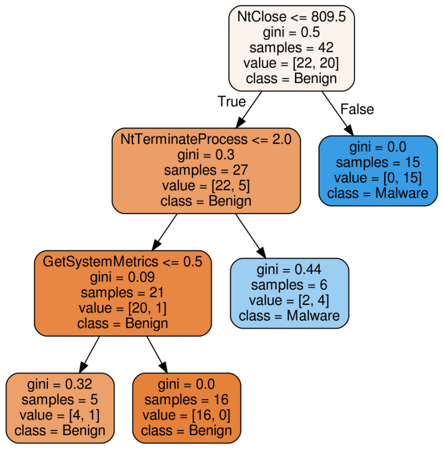


Figure 4.3: Decision Tree for Small Sample

4.3.3 Discussion and Recommendations

Our research shows that the analysis of the varying frequencies of Windows API calls made between malicious and benign software can be used for classification with extremely high accuracy depending on the machine algorithm used and characteristics of the dataset. Our Large Sample dataset (reported 99% accuracy) trained the models to recognize the most malicious activity from the features because the data was highly biased toward understanding the Windows API call frequencies as malicious behavior. Our Small Sample dataset (reported 88% accuracy) contained an equal number of malicious and benign software samples, resulting in a bias toward understanding the feature properties as benign indicators – likely due to a higher number of benign samples in the training dataset for the model not producing enough distinguishing features between the classes. Using the Goldilocks analogy, the Medium Sample dataset (reported 96% accuracy) was a "just right" middle-ground which captured more true negatives and true positives for malware classification based solely on Windows API call frequency.

Our research also shows that the malware analysis machine learning models generated by various algorithms are subject to varying accuracy depending on the data used to build those models as well as the algorithms selected. For example, Naive-Bayes did not produce strong models for us, possibly due to the conditional independence assumption inherent in the algorithm. Furthermore, decision trees trained on our dataset using entropy to determine the root note were much more accurate than those using Gini. Indeed, our research suggests that there is no “one size fits all” for malware detection using machine learning. As noted previously, recent work has shown the efficacy of using machine learning algorithms to identify malware based upon observed behavior through dynamic analysis. Our research supports the viability of this approach for malware detection, with the added caveat that it is simple to overfit or underfit machine learning models to the malware behavior data. Considering that there is currently no universal dataset for malware behavior or a recognized unbiased approach to creating models in a consistent manner, we find that it is imperative for researchers to employ exhaustive investigation into varied machine learning algorithms and techniques when evaluating the efficacy of a model’s ability to identify malware so as to not fall victim to the bias of a given dataset or methodology.

We also believe it is worthwhile to consider not simply the accuracy score of a machine learning algorithm but also what meaning can be derived from the produced model. Decision trees in particular give us information about the relation between features when observed visually. The differences in the trees described in Figures 4.1, 4.2, and 4.3 provide an opportunity to analyze the logic employed by the machine learning algorithm in developing the model. As the model learns to understand malicious behavior, we have discovered the possibility of using the relationship between feature values as an indication of how to understand what patterns exist in the dynamic analysis of malware. This suggests that there is potential for fingerprinting dynamically observed malicious activity based upon common Windows API calls made which perform actions that are typically malicious in nature because they serve a typically malicious purpose. This is a different approach from recognizing the frequencies of API calls made by malware in that we would not be concerned with just one or more function calls - our concern would be in what these function calls tell us about the malicious or benign intent of the software.

Chapter 5

Malware Family Fingerprinting Through Behavioral Analysis

In Chapter 4 we showed how machine learning using operating system Application Programming Interface (API) calls made by software as an input can be made to discern malicious from benign software. In the present chapter, we show how this machine learning technique can be extended to accurately fingerprint malware families. This allows for greater intelligence to be gathered from a malware executable in an automated fashion which requires no expert knowledge of malware in order to be effective.

5.1 Introduction

Malware signatures are devised by Anti-Virus (AV) software vendors which allow for the recognition of known malware, leading to the deletion or quarantine of the malicious program before harm can be performed. The challenge AV faces in terms of efficacy lies in that these malware signatures are only capable of detecting known malware and offer little protection for new variants. AV software vendors provide frequent updates on their malware signature databases for systems which have the AV software installed, but this methodology can not help increase the security of a system which has already been compromised by an unknown malware prior to the signature database update. This inherent flaw in AV security provoked the creation of intrusion detection and prevention systems and other cybersecurity defense tools which add layers of security to help protect computing systems from malware compromise.

There is no standard for the creation of malware signatures. The many Anti-Virus software companies work independently to devise malware signature databases in competition, with each attempting to produce a superior product to sell. In doing so, malware researchers for each AV company analyze known malware programs to find elements common to malware types, so as to maximize the rate of detecting malware while attempting to minimize the false-positive rate. False-positive malware detection involves the inaccurate identification of benign software as malicious. A high false-positive rate would in turn lower the confidence in the AV software as normal business functions on a computer can be disrupted due to such an error. This leads to the creation of AV signatures which attempt to detect specific and often granular types of known malicious code or behavior in attempt to increase accuracy.

Malware signatures are associated not only with malware types such as a virus, trojan, or worm, but also by a classification of sets of malware which can be referred to as families. Malware families include a certain strain of malware and its derivatives. Just as AV software companies invest effort into creating better signatures and AV software engines, malware authors continue to devise methods to obfuscate malicious code and avoid detection. Polymorphic malware variants are one such method used to evade AV detection, where the code of the malware executable allows for self-replication along with a re-ordering or restructuring of the code itself. Such altered malicious programs may perform the same actions upon a victim computer regardless of the structure of their code and it are these actions which help to define the malware family classification.

One benefit of anti-virus signatures for the malware researcher is the label of a malware type associated with the signature. This allows one to group many unlabeled malware samples by the corresponding signature for a certain type of malware. In this fashion, malware behavioral analysis can be performed upon similarly labeled malware samples in order to examine what similarities exist in the malware of the certain classification. In our previous work we have shown how Windows API system function calls can be used to perform behavioral analysis on malware, which in turn allows for the categorization of malware via multiple machine learning classifiers based on the frequencies of Windows API calls invoked by the analyzed malware [76][77]. In the present work we demonstrate that behavioral analysis of Windows API function calls made by sets of malware classified as a certain type through AV signatures results in stacked ensemble machine learning models capable of discerning malicious software as a certain malware family classification. In the process, we also describe what behaviors or attributes of the malicious software which are common amongst the malware family and in essence reverse-engineer the malware signature used to classify the malware.

5.2 Machine Learning Enabled Malware Family Classification

In this section we describe our methodology and results from applying a stacked ensemble machine learning approach to malware family classification.

5.2.1 Setup & Methodology

We utilized the Cuckoo Malware Sandbox to produce reports for the behavior of over 65,000 malware samples. Cuckoo delivers these reports in JSON format, which allows for convenient parsing of the relevant Windows API function call data. The name of each API utilized along with the number of times each API was called for each malware file was recorded, as our focus in this research is API call frequency.

Additionally, we wanted to label each malware file based upon anti-virus signature detection. For this purpose, we used VirusTotal[56], an online tool which allows multiple anti-virus software products to analyze malware and reports any detections. Microsoft Defender Antivirus consistently provided malware detections and classification for each malware in our set, so we decided to use their classification for our malware family labeling. Not every other anti-virus vendor provided such labeling for every malware sample, so our choice of using Microsoft's classification was based on availability of data for known malware samples.

Table 5.1: Top Twenty Families in Malware Set

Malware Family	Accuracy
PUA:Win32/DownloadGuide	99.5539%
PUA:Win32/Presenoker	99.4711%
Trojan:Win32/Occamy.C	99.4667%
SoftwareBundler:Win32/Prepscram	99.4194%
Trojan:JS/Redirector.QE	99.3849%
Virus:HTML/Jadtre.A	99.3371%
TrojanDownloader:JS/Vigorf.A	99.2945%
TrojanClicker:JS/Faceliker.D	98.9730%
Trojan:HTML/Redirector.CF	98.8959%
Trojan:HTML/Redirector	98.6964%
Trojan:HTML/Brocoiner.N!lib	98.6839%
TrojanClicker:JS/Faceliker.H	98.5605%
Trojan:JS/Redirector	98.2740%
Trojan:JS/HideLink.A	98.0905%
Virus:VBS/Ramnit.gen!C	95.4734%
Trojan:JS/Iframe.AE	94.2849%
Trojan:HTML/Brocoiner.A!lib	93.0102%
TrojanDownloader:JS/FakejQuery.AR!MTB	91.7861%
TrojanDownloader:JS/FakejQuery.A!bit	90.1614%
Virus:VBS/Ramnit.gen!A	79.4957%

A CSV file was created which represented our malware dataset, with each row describing a particular malware file and each column describing the frequency of a certain

API call. Separate CSV files were created for the top 20 most prevalent of the 318 identified malware families, where the rows of each malware corresponding to a particular malware family were labeled in a fashion which supports supervised machine learning. Measures were taken to adjust the ratio of labelled malware samples for each CSV file in order to combat over-fitting or under-fitting the data. We then created a stacked ensemble machine learning model using scikit-learn in Python, making use of KNN, logistic regression, and decision tree classifiers to distinguish the various malware families according to API frequency relationships.

5.2.2 Machine Learning Results

The machine learning models used were collectively able to positively classify each malware file with 96.5156% accuracy. This demonstrates that API call frequency is a valid predictor of malware family. We believe the accuracy could be improved by further increasing the number of malware files in the dataset, which should serve to likewise increase the diversity of malware family members.

Of interest was the accuracy of the model to detect "Ramnit.gen!A"[78] which performed at 79.4957% accuracy, the poorest of the lot. In contrast, the related malware "Ramnit.gen!C"[79] was classified with 95.4734% accuracy based on API call frequency. We will analyze the cause for this in the next section.

5.3 Malware Signature Analysis

The particular label for a malware which matches a signature will vary amongst the different AV software companies. For example, Microsoft Defender Antivirus created a signature for a malicious software they named "TrojanDownloader:JS/Vigorf.A"[80] which Trend Micro describes as "Worm.JS.Bondat.AC"[81]. Evidently, there is cause for debate for whether a particular malware is a trojan or a worm. While we cannot explain the methodology taken by malware analysts for different anti-virus companies in terms of the classification of malware, we can show that many discrepancies exist between how the same malware file will be classified by different anti-virus software signatures. In the general case, these discrepancies may be trivial for as long as the end result is the deletion or quarantine of the malicious file is accomplished before a victim computing system is compromised.

Through behavioral analysis we were able to discern key characteristics between malware classified as a certain type through our means of signature labelling. Malware labeled as Vigorf.A attempt to communicate to a particular URL in attempt to download a file. Additionally, specific character strings exist in the malware which differentiate it from other malicious software. Similarly, malware labeled as "Trojan:JS/Redirector.QE" attempts to call out to a range of URLs in attempt to download a specific file named "jquery.min.php." Since this file name is hard coded in the code instruction, this can be used for string comparison to detect similar malware. Other malware files labelled as "Jadtre.A" display an attempt to reach out to a specific URL in attempt to download additional malware. Similar specific URL or file name specifications are prevalent in many of the malware families we analyzed, leading us to believe that many of the anti-virus signatures we employed for family classification utilize a simple string-matching method.

For our purposes we desire a means of malware family classification through the analysis of Windows API call frequencies. In the previous section we described a machine learning model which accomplished this with high accuracy. In this section we will describe why this relationship between API calls and malware families exists.

5.3.1 Windows API Frequency Analysis

When we consider that the operations which software, malicious or benign, are programmed to perform on a system must do so through API calls to the host operating system, we can understand that the frequency of these calls produces a fingerprint. This fingerprint can be used to identify software which has been developed to perform similar actions. For example, Figure 5.1 shows the frequencies of API calls made by multiple files which matched the Microsoft malware signature for "DownloadGuide." Each column represents a behavioral analysis report for an individual malicious software file while each color represents a specific API call, with the size of the colored sections representative of the relative API call frequencies.

We can visually determine a pattern in these malware files by comparing the frequencies of various API calls. This is the means by which the machine learning models also determine malware family relationships. It is interesting to note that the number of unique API calls made by malware does not appear to significantly affect the accuracy of the malware family categorization. For instance, the DownloadGuide malware shown in Figure 5.1 was observed making use of 152 unique API calls with certain frequencies in our malware set, while the Redirector.QE malware shown in Figure 5.2 called out to only 24 unique API functions. Our experiments show that it is not necessarily which API calls were chosen by malware which provides a means of fingerprinting, but the relative frequencies of these API calls made which provides the ability to discern what family a certain malware is most associated with.

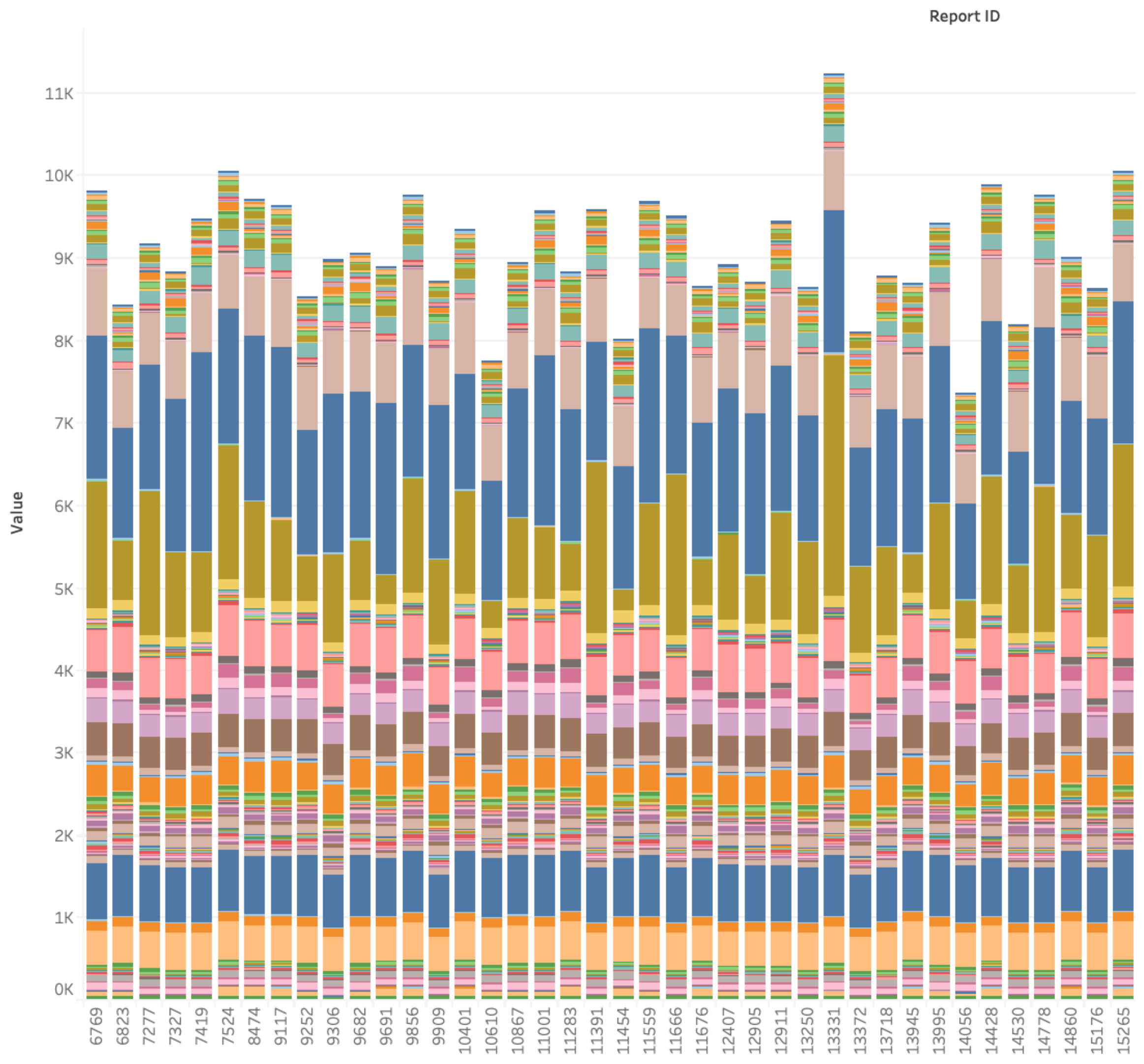


Figure 5.1: API Frequency for PUA:Win32/DownloadGuide

5.3.2 Limitations of Signature-Based Approach

In Section III we discussed the results of our machine learning models and how of the twenty most prevalent malware families in our malware dataset, each model was capable of identifying the correct malware family with very high accuracy with the greatest

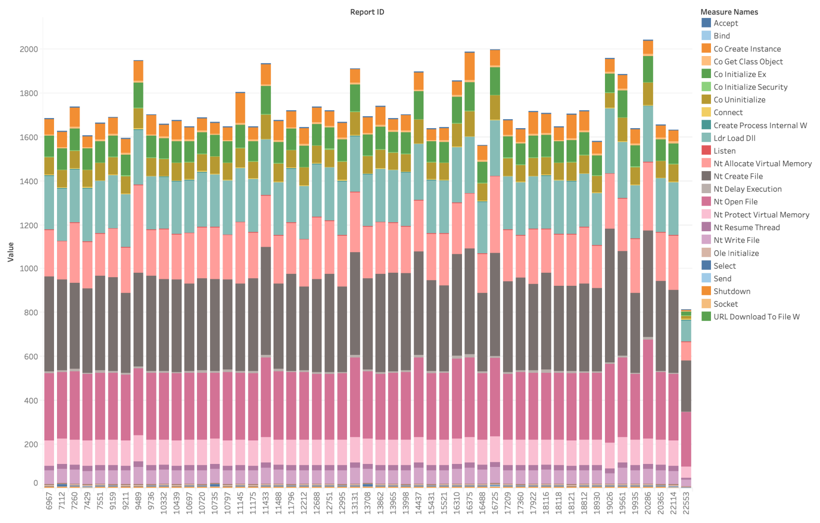


FIGURE 5.2: API Frequency for Trojan:JS/Redirector.QE

exception being the model trained to identify malware classified as Ramnit.gen!A. The similar malware family Ramnit.gen!C performed with 15.9777% greater accuracy. We found the question of why there would be such a discrepancy between the accuracy of these two related families to be intriguing.

Behavioral analysis of malware labeled as these two forms of the Ramnit virus reveals that malware classified as Ramnit.gen!C in our malware set make use of 123 unique API function calls while malware classified as Ramnit.gen!A utilized 287 unique API calls as a whole. Additionally, the frequencies of these API calls are much more uniform in the set of Ramnit.gen!C malware than in the Ramnit.gen!A set. Table II describes the twenty most prevalent API calls in the Ramnit.gen!A malware set, with Table III similarly describing Ramnit.gen!C. We believe that this lack of uniform API frequency contributed to the lower accuracy of the Ramnit.gen!A machine learning classifier.

Table 5.2: Top Twenty APIs in Ramnit.gen!A Malware Set

API Function	# of Calls
RegQueryValueExW	92027468
RegOpenKeyExW	32498784
NtClose	28061847
RegCloseKey	23596025
GetSystemMetrics	22887494
NtWriteFile	16457483
NtCreateFile	16100748
NtAllocateVirtualMemory	11930672
NtOpenFile	9044230
NtQueryDirectoryFile	7218844
LdrLoadDll	6826873
GetSystemTimeAsFileTime	6345445
LdrGetProcedureAddress	6194868
RegEnumKeyW	5784563
FindFirstFileExW	4927071
NtOpenKey	4806287
SetErrorMode	4756149
CoInitializeEx	4664570
NtProtectVirtualMemory	4654408
RegEnumValueW	4574213

Microsoft's documentation on these two viruses is unclear on the technical specification for their classification. However, we do find that both Ramnit.gen!A and Ramnit.gen!C perform similar actions on a victim computer, including an attempt to drop a malicious file named "svchost.exe" to the logged-in user's temporary directory through the use of malicious Visual Basic Script code embedded in an HTML file. However, different malware files labeled as Ramnit.gen!A may perform many additional functions, such as the modification of the Windows registry and/or the creation of additional files to be run as DLLs. This wider range of possible activities appears to suggest that the Ramnit.gen!A signature may be over-broad and has the ability to mis-label related yet different-generation malware.

This illustrates the importance of accurate malware signatures for the purpose of classifying related malware behavior. We find that many malware families classified by Microsoft AV have a range of different classifications according to other AV software companies. Future work may benefit from examining malware family classification based on malware signatures produced by a variety of AV software. In so doing, the possibility of mis-labeling malware by family may be reduced and possibly new malware families can be distinguished by comparing multiple signatures with behavioral analysis.

Table 5.3: Top Twenty APIs in Ramnit.gen!C Malware Set

API Function	# of Calls
NtWriteFile	10586688
NtCreateFile	626186
NtOpenFile	568450
NtAllocateVirtualMemory	430983
LdrLoadDll	363739
NtProtectVirtualMemory	190428
CoInitializeEx	155245
CoUninitialize	128127
CoCreateInstance	98541
NtResumeThread	35054
RegQueryValueExW	22989
CoGetClassObject	14364
OleInitialize	9811
NtDelayExecution	9549
RegOpenKeyExW	8070
NtClose	6404
RegCloseKey	5848
GetSystemMetrics	5704
send	5064
CoInitializeSecurity	4262

5.4 Discussion and Recommendations

In this chapter we showed how behavioral analysis of malware combined with a data set of Windows API function call frequencies classified by anti-virus signature matches allows for the creation of machine learning models capable of identifying the appropriate family to which a given malicious software belongs. We described how API frequencies can be used for this classification and explain that this form of software behavior fingerprinting can be used with very high accuracy. Finally, we discussed how this current methodology is dependent upon accurate anti-virus signatures.

We believe that this methodology can be extended by including additional anti-virus signatures from multiple sources as a means of increasing accuracy. Furthermore, we believe that comparison of known malware families and generational changes through API frequency analysis may provide insight into how malware evolves over time, leading to a potential predictive model for future malware variants based on historical data. In so doing, we believe it is possible to determine a kind of "natural selection" for successful malware families and thereby contribute to cybersecurity defense.

Chapter 6

Friend or Foe: Discerning Benign vs Malicious Software and Malware Family

In Chapter 5 we showed how machine learning can be used to fingerprint malware families. This process, while effective, comes at a cost of time and resources. We wished to develop a means of performing similar actions with greater speed. This led us to consider malware and malware family fingerprinting using only a fraction of the total API function calls performed by the software being examined. This chapter presents the results of this research, which compares MLP, CNN, and SVM machine learning algorithms.

6.1 Introduction

Malware, or malicious software, continues to threaten computer systems and networks ranging from the home office to corporate environments, including workstations, mobile devices, and Internet-of-Things. Malware continues to be successful because it is just as varied as the systems and users they intend to compromise. As world events such as technological innovations or a global pandemic alter the way in which we use computing devices, threats from malicious software continue to diversify, thereby continuing to frustrate cybersecurity professionals who seek to ensure the safety of the systems and networks they secure.

The threat of a malware-based compromise is particularly severe for enterprise environments which rely heavily upon Microsoft Windows systems. Malware compromises continue to rank higher in frequency for Windows systems than any other operating system [67], due in part to the prevalence of Windows systems located both in the enterprise as well as home environments. Malware-based compromises can result in a ransomware attack, remote access to the compromised system, data theft, or other forms of abuse by the malware author or remote attacker.

Many tools currently exist to aid in the fight against malware, including anti-virus or anti-malware software. However, these tools are limited to known malware signatures. Endpoint threat detection software is an advancement in the fight against malware yet is mostly effective against known threats. Next-generation firewalls add intrusion prevention to their list of offerings while also relying on signatures and rules defined in advance of new, unknown threats. This produces a landscape of tools and methodologies which require significant investment in terms of both money and personnel to use them. As the ways in which users interact with business networks and applications change, the need for a dynamic approach to cybersecurity is not always reflected in the security tools made available, especially to those on a limited budget or time.

This has motivated us to develop a framework for an approach to not only identify malware from benign software, but also to discern malware family to provide greater threat intelligence. Both malicious and benign software perform many, many operations on a system when they are executed. Analysis of hundreds of thousands of behaviors performed on a system can be costly, time consuming, and may require advanced cybersecurity training. However, machine learning provides the opportunity to automate these tasks and produce more legible results, especially for organizations which lack a dedicated team of skilled human analysts.

6.2 Malware Behavior Analysis

In order to programmatically observe the behavior of malware in an isolated setting, we designed an environment to allow for the installation of Cuckoo and the analysis of known malware samples in a virtual machine sandbox per the installation instructions provided by Cuckoo[22]. Our goal was to focus on the evaluation of potentially

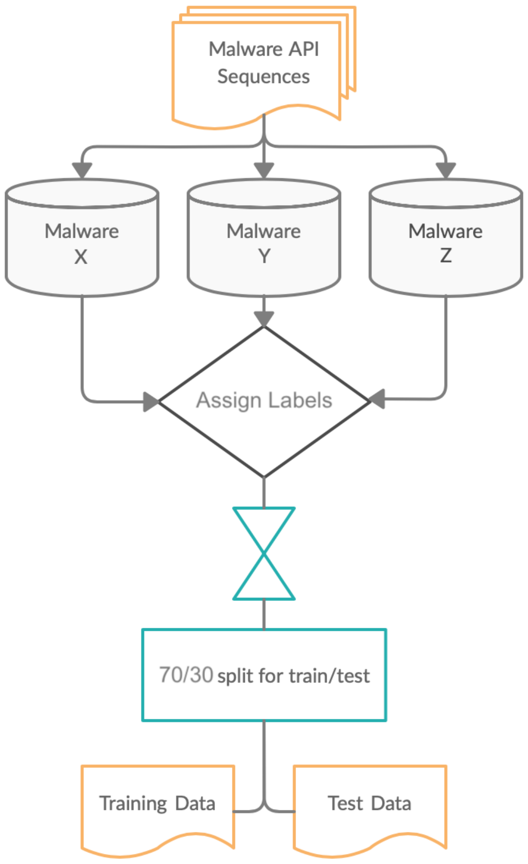


FIGURE 6.1: Malware Data Preparation

malicious software affecting Windows operating systems, and the ability to discern between malware types, families, and benign software. The decision was made to focus on malware affecting Windows systems due to the significantly larger amount of malware compromises observed on Windows systems as compared to other operating systems [67], and the relative threat such malware presents to corporate environments.

6.2.1 Setup and Malware Dataset

Cuckoo was configured per the installation guide found on the Cuckoo website[22], including two 64-bit Windows 7 virtual machines installed on the Ubuntu host. Cuckoo supports many virtualization software solutions but does assume the usage of VirtualBox[82] by default, so for ease of setup we chose this platform. VirtualBox is a free system virtualization product developed by Oracle and it easily integrates with Cuckoo for administration of the virtual machines.

Known malware samples were acquired from VirusShare [83], an online resource of malicious software containing multiple versions of malware samples seen over time. This allows for the observation of evolving behaviors as the methods of exploiting system and application vulnerabilities changes with new generations of malware.

6.2.2 API Collection Methodology

Once the analysis has been performed, Cuckoo generates a report of the observed activity including, but not limited to, changes to the registry, newly spawned processes, file creation and access, virtual memory access, HTTP communication to an external IP, and much more. When a malware file is analyzed by Cuckoo, a report is generated including a list of each behavior exhibited by the malware as it was executed in a controlled environment. This report can be delivered in JSON format, which can easily be parsed for the relevant behavior artifacts. These artifacts are expressed through the Windows API[71] system function calls made by the malware to affect the system during execution. To make use of this information, we gathered the names of the first 3,000 API function calls for each malware analyzed. We found that less than this number of API calls negatively affected the overall accuracy of our machine learning models, while a greater amount provided only little improvement to accuracy at the cost of a much greater training time. We also analyzed several commonly available software executables with Cuckoo to obtain API call behavior for comparing malicious software against benign software.

6.2.3 Malware Classification

The collection of malware available from VirusShare was delivered without any identifiers. This meant that while it was known that the supplied malware samples were malicious, there were no labels to identify the malware by type (virus, trojan, worm, etc.) or by family (Ramnit, Faceliker, Brocoiner, etc.). As we were very interested in discovering any similarities between malware types, we felt that identification of our malware samples was important. VirusTotal[56] is an online resource which allows for the comparison of the hash of each unique malware file to any malware identified by various antivirus vendors. Each of the antivirus vendors who reported that a given malware file was indeed malicious assigned a name for the family to which the malware belonged. These malware family names vary as there is no set standard nomenclature. We chose to follow the naming convention used by Microsoft antivirus, since a version of their antivirus software is available on all modern Windows operating systems, making this an appropriate baseline. As shown in Figure 6.1, the labeled API calls made by each malware were then collected according to their family label.

These Microsoft antivirus signatures were referenced to assign a malware family to each malware file. For the current research data set, this resulted in 23 family designations for the set of unique malware. We then labeled representative malware samples by these family names and collected the full Windows system API function calls for each malware file. For ease of reference, we assigned unique identifiers to each malware as described in Table 6.1.

6.2.4 Benign Software API Collection

In addition to behavioral analysis of malware, we also collected Windows API calls for known benign software. This was performed to compare the behavior of benign software to malware for the purpose of showing that our machine learning methodology can discern benign activity through API analysis. The choice of what benign software to use was arbitrary, with a tendency to collect software that might be commonly downloaded by common computer users. Each of the benign software files were executed in the Cuckoo sandbox and API calls were observed in the same manner as for the malware analysis. The benign software executables as well as their reference labels are described in Table 6.1.

Table 6.1: Malware by Antivirus Signature Classification & Benign Software Set

ID	Signature	ID	Benign Executable
M1	Virus:VBS/Ramnit.gen!A	B1	7-Zip 32-bit[84]
M2	Virus:VBS/Ramnit.gen!C	B2	7-Zip 64-bit[84]
M3	PUA:Win32/Puamson.A!ml	B3	Avira Antivirus[85]
M4	TrojanClicker:JS/Faceliker.M	B4	CCleaner[86]
M5	Trojan:JS/Iframeinject	B5	Google Chrome[87]
M6	Trojan:HTML/Redirector.CF	B6	Epson scanner software[88]
M7	Trojan:Win32/Skeeyah.A!bit		
M8	Exploit:HTML/IframeRef.gen	B7	GifCam animated gif software[89]
M9	Virus:VBS/Ramnit.B		
M10	TrojanClicker:JS/Faceliker.D	B8	GIMP image editor[90]
M11	TrojanClicker:JS/Faceliker.C		
M12	Trojan:JS/Redirector.QE	B9	OpenVPN[91]
M13	Trojan:JS/BlacoleRef	B10	Ultrasurf proxy[92]
M14	PUA:Win32/Presenoker	B11	Microsoft Visual Studio Code[93]
M15	Trojan:HTML/Brocoiner.D!lib		
M16	TrojanClicker:JS/Faceliker!rfn		
M17	Trojan:Win32/Vibem.O		
M18	Trojan:HTML/Redirector.EP		
M19	Exploit:HTML/IframeRef		
M20	TrojanClicker:JS/Faceliker.A		
M21	Exploit:HTML/IframeRef.DM		
M22	Trojan:HTML/Phish		
M23	PUA:Win32/Kuaiba		

6.3 Machine Learning Framework

Our dataset of malicious and benign Windows API calls provided a description of the behavior each software performed upon the Windows system in our Cuckoo sandbox environment. Frequency analysis of these APIs for malware classification has been performed in the past [77], however we felt that the goal of our experimentation should be to discover what relationships might be observed between malicious and benign software through inspection of these API calls. To that end, we decided to make use of machine learning for analysis of API calls made by our software dataset to determine if this data was suitable for the task. We chose to use the machine learning algorithms described below to devise models which compare three sets of input data for multi-class classification. For each experiment, API calls for three software executables (malicious or benign) were used for input into these models. The APIs were input into the algorithms one at a time, so the API function names are considered in chronological order by label.

6.3.1 Machine Learning Algorithms Used

We chose to implement three machine learning algorithms for our experimentation, namely Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), and Support Vector Machine (SVM). These models were chosen for their applicability to classification prediction problems and accuracy performance.

6.3.1.1 Multilayer Perceptron

We built a learning network using MLP as shown in Figure 6.2. This consisted of five dense layers with input dropout of 40%, compiled with a categorical cross-entropy loss function and the Adam optimizer. As noted previously, we found the greatest success in our experimentation when using a network created through MLP. Therefore, the analysis in Section V reflects the results from our MLP learning model.

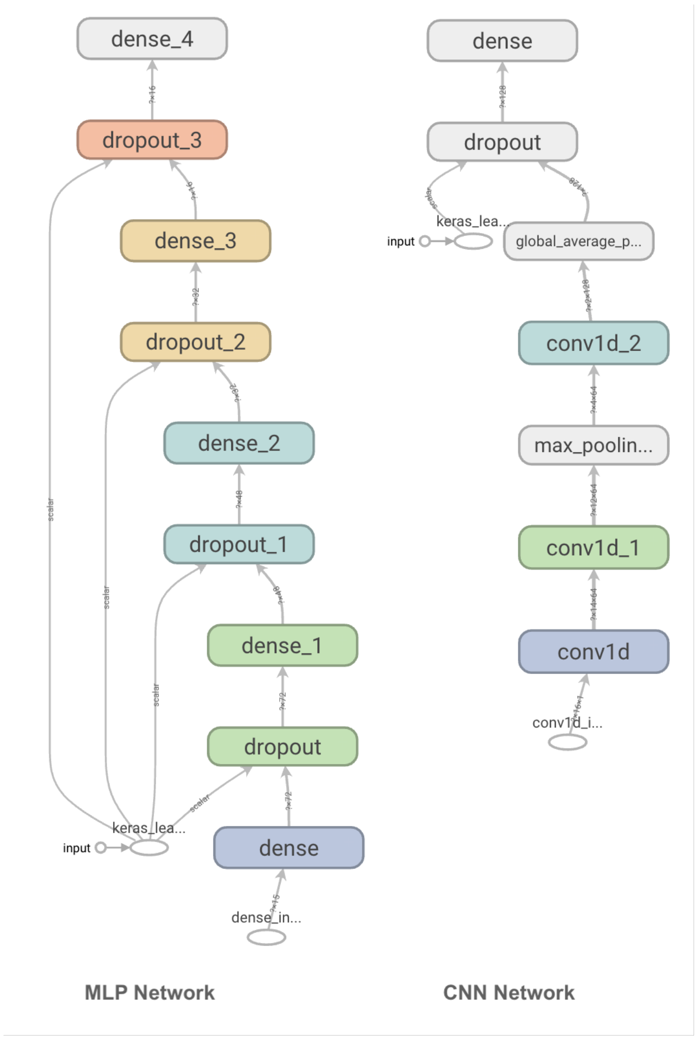


FIGURE 6.2: MLP & CNN Network Graphs

6.3.1.2 Convolutional Neural Network

We built a learning network using CNN as shown in Figure 6.2. This consisted of three convolutional layers with an input dropout of 50%, compiled with a categorical cross-entropy loss function and the Adam optimizer.

6.3.1.3 Support Vector Machine

Finally, we built a learning network using SVM. We used the linear kernel function with the one-versus-one function for multi-class classification.

6.3.1.4 Comparison of Algorithms

We experimented with multiple ratios of training and testing data, including 80/20, 70/30, 60/40, and 50/50, respectively. We found that in general the best performance for these machine learning algorithms for our data came with a 70/30 split between training and test data sets.

A comparison of the accuracy of these machine learning algorithms with respect to our data is shown in Figure 6.3. Here it is shown that the overall accuracy of the MLP algorithm performs better than CNN or SVM for each experiment. For example, experiment 1.3 resulted in an overall accuracy of 88% when MLP was used, as opposed to accuracies of 50% and 54% for CNN and SVM, respectively (more details regarding experiments and experiment numbers are provided in the next section). The only experiments where MLP did not perform better than the other algorithms were in the cases of experiment 3.3 where CNN was 2% more accurate and experiment 7.2 where both CNN and SVM were 1% more accurate. Given these small values of difference in accuracy while MLP excelled in all other experiments, we refer solely to the experimental results from usage of the MLP model in subsequent sections.

We feel it is important to try to explain why these algorithms behaved so differently. First, MLP maps the features from the input space to the output space – meaning that it takes input and then adjusts the weights during the optimization such that an optimal function is generated. This creates a more robust mapping from the input to the output so that the relationship can be learned with greater reliability when optimized. MLP can efficiently be used with a limited number of features, such as with our problem. Since our problem uses few features, MLP maps our functions from input to output spaces with the high accuracy.

CNN operates by taking convolutions of the input. CNN typically extracts a very low level of information from the input and thus helps in the classification task. The convolution and pooling layers in the CNN are typically used to extract features from image data, although they are also useful for one-dimensional data when spatial information is of interest. However, there is no spatial information in our dataset and therefore CNN probably is not performing better than MLP because of limited low-level data extraction.

SVM finds hyperplanes in a n-dimensional space, where the different classes are separated by the boundaries of the resulting planes. If the different classes are well separated, then SVM can be very effective as the defined boundaries can easily classify the dataset. It is probable that our dataset does not have well separated classes. Therefore, SVM does not work as effectively and more complex mapping between input and output using MLP performs better.

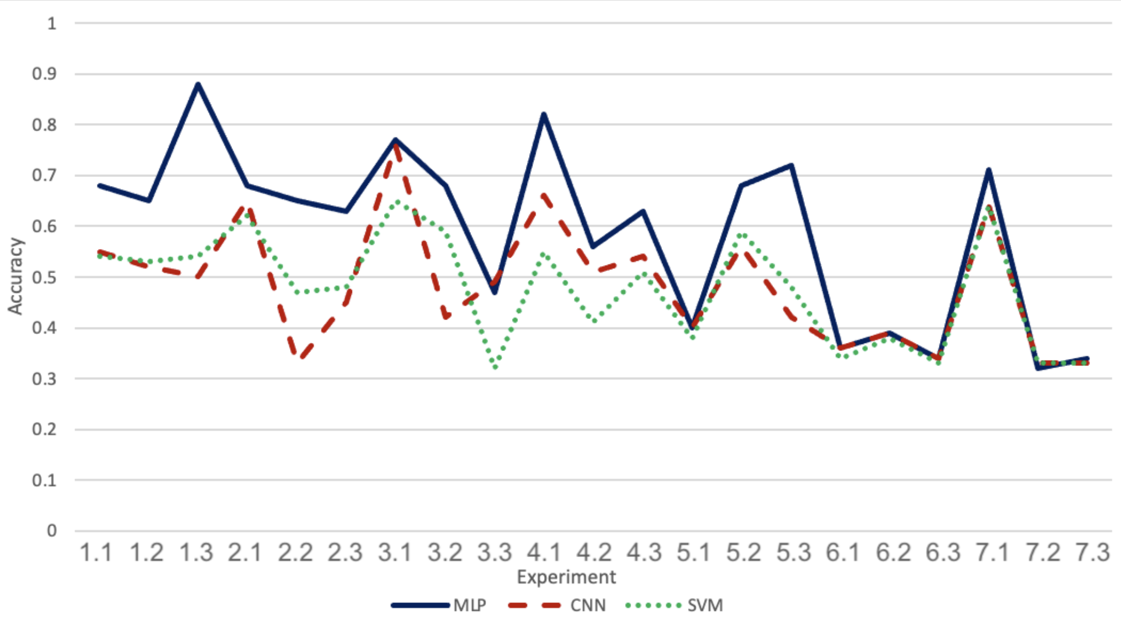


FIGURE 6.3: Comparison of MLP Algorithm Accuracy

Overall, our MLP networks performed with greater accuracy than our CNN or SVM networks. As a result, we have included only the MLP results in this work.

In addition to its increased overall accuracy with our Windows API dataset, our MLP approach performed much faster than traditional CNN by a significant amount. Three days were required for training and testing each experiment using our CNN network, on average. By comparison, our MLP network required an average of four hours per experiment.

6.3.2 Experimentation

Seven experiments were devised for testing the machine learning models described in the previous section. This included the following:

1. Benign vs Trojan vs Virus
2. Benign vs Trojan vs Trojan
3. Benign vs Benign vs Benign
4. Benign vs Benign vs Malware
5. Trojan vs Virus vs PUA
6. Trojan vs Trojan vs Trojan
7. Related Malware (by signature)

These experiments were purposefully chosen to assess the ability of our MLP networks to discern malware families as well as benign software from a variety of input sets of API data. Each experiment consisted of three tests, where each test concerned sets of input API calls from three different software sources. Therefore, the experiments are referred to as 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, and so on through 7.1, 7.2, and 7.3.

For example, our first experiment involved the input of a benign software, a trojan, and a virus. Our third test within the first experiment is then referred to as 1.3 and involved the benign Avira Antivirus (B3), the trojan Win32/Vibem.O (M17), and the virus VBS/Ramnit.B (M9). The results from our MLP network are recorded in Table 6.2.

Similarly, our seventh experiment involved the input of three related malwares. Test 7.1 involved APIs observed from three variants of the Ramnit virus, including VBS/Ramnit.gen!A (M1), VBS/Ramnit.gen!B (M9), and VBS/Ramnit.gen!C (M2). The results from our MLP network are recorded in Table 6.3.

Extensive results from our experimentation can be found in Appendix A.

TABLE 6.2: Statistics for Experiment 1.3

	Precision	Recall	F1-score	Support
M17	0.98	0.78	0.87	880
B3	0.83	0.86	0.84	914
M9	0.85	1.00	0.92	906
Accuracy			0.88	2700
Macro Avg.	0.89	0.88	0.88	2700
Weighted Avg.	0.89	0.88	0.88	2700

TABLE 6.3: Statistics for Experiment 7.1

	Precision	Recall	F1-score	Support
M1	1.00	0.96	0.98	913
M9	0.55	0.74	0.63	882
M2	0.61	0.42	0.50	905
Accuracy			0.71	2700
Macro Avg.	0.72	0.71	0.70	2700
Weighted Avg.	0.72	0.71	0.70	2700

6.4 Analysis of Results

Our initial experimentation included the experiments 5.1 - 7.3, including the classification of three sets of trojans, viruses, and potentially unwanted programs (PUA), three sets of trojans of different malware families, and three sets of related malware families. We found success in our machine learning model's ability to discriminate between different malware types, as can particularly be seen in Table 6.2. These experiments show that there is enough of a difference in the API calls made by the malware Trojan:Win32/Vibem.O and Virus:VBS/Ramnit.B in particular to make this methodology useful for learning the type of malware being analyzed. In this way, we show that classification of malware by family is possible through observation of API function calls to learn distinct patterns.

We also saw success in the discrimination of malware of related families - in particular, our model could discern the difference between Virus:VBS/Ramnit.gen!A, Virus:VBS/Ramnit.B, and Virus:VBS/Ramnit.gen!C with high accuracy in experiment 7.1 and shown in Table 6.3. However, we were not quite as successful in experiments 7.2 and 7.3 concerning Faceliker and iframeRef variants, respectively. We believe this is because there is more of a functional difference between the Ramnit variants, as analysis of the malware behavior shows differences in how these viruses compromise a victim computer. This suggests that our methodology is useful for determining differences in actions between malware variants of the same family and could be helpful for fingerprinting malware evolution.

Expanding our scope to include benign software in experiments 1.1 - 4.3 allowed us to determine what impact was provided by including benign software in our learning methodology. We found a much higher average accuracy for these experiments, compared to the experiments concerning only malware. We believe that the variety of benign software used in our experimentation was varied enough in function to not reflect a distinct difference in behavior from malware, as both the benign and malicious software used perform similar functions on a Windows system (file read/writes, registry changes, creation of processes to inject code, etc.). However, we found interesting results when comparing similar experiments which were differentiated by the inclusion of benign software.

For example, experiment 4.1 involved three malware inputs - namely, Virus:VBS/Ramnit.B, Trojan:JS/Redirector.QE, and PUA:Win32/Puamson.A!ml. Experiment 1.1 replaced PUA:Win32/Puamson.A!ml with the 32-bit 7-Zip executable which resulted in a change from an overall accuracy of 40% to 68%. We believe that this increase in accuracy is a result of the variance in API calls made by benign software as opposed to malware. Experiments 3.1 - 3.3 include only benign software inputs while experiments 6.1 - 6.3 include only trojan malware inputs and the former reports a greater ability to classify over the latter. This suggests that the API calls for malware lack the entropy found in benign software API calls. This would then possibly explain the greater ability to differentiate malicious from benign software over malware of the same type.

These results show that our approach provides a means of understanding the nature of malware by learning the behavior of different malware types and family relationships. This benefits the cybersecurity incident responder by providing an additional means of malware analysis, where the relative risk presented by a certain malware not matching a current antivirus signature can be assessed by its behavioral relationship to known malware families.

6.5 Discussion and Recommendations

Our research showed that it was possible to discern malware and benign software through learning the Windows system API function calls made by different classifications of software. The novelty of this approach can be found in how accurately it performed, given the sparse input of single API function call names. This produced a framework for quickly learning the differences between software to accurately predict, not only if a given software is malicious or benign, but also to classify malicious software by family type. The accuracy of this approach increased when including disparate software types and we believe that the overall, general accuracy will be increased by adding additional classes with a mix of benign and malicious software. While not a replacement for current malware detection mechanisms, this approach supplied a quick tool for accurate malware analysis as part of a cybersecurity incident response process to provide greater insight and visibility into the nature of malware. One that, otherwise, may be unavailable for many cybersecurity professionals.

Chapter 7

Ohana Means Family: Malware Family Classification using Extreme Learning Machines

In Chapter 6 we compared several machine learning algorithms for greatest accuracy when learning the differences between malware, benign software, and malware family. An additional problem we discovered was that while we were able to identify MLP as a fast algorithm suited to making this discrimination based upon single API function calls, this methodology did little for malware family classification using longer sequences of API calls. The motivation for this research included understanding how the speed and accuracy of malware family prediction would be affected by sequences of varying sizes. Additionally, we were interested in the use of the Extreme Learning

Machine (ELM) algorithm to see if its greater speed would be appropriate with our malware data set.

7.1 Introduction

Malicious software, or malware, continues to be a grave threat to computing systems, especially those running the Windows Operating System in enterprise environments[94]. It is the role of cybersecurity incident responders to detect cyber threats such as malware compromises and to quickly remediate infected systems before damage can be done. However, this task is not trivial - in the case of one type of malware called ransomware, \$11.5 billion in damages were accrued in 2019 in the United States alone[95]. The risk associated with malware builds a need in the cybersecurity industry for real-time malware detection which can continuously learn and better classify malware threats.

Traditional antivirus (AV) software is one form of malware detection, where software is compared to known malware code components, known as malware signatures. If there is a match between a malware signature and the inspected software, the AV software may quarantine or delete the offending program. This process can be effective; however, its efficiency is predicated upon the prior knowledge of the malware threat. A certain malware must already have been detected and analyzed by the AV vendor for the malware signature to exist. This limits the efficacy of AV software to only known malware threats. Unfortunately, this provides little protection against new malware variants which lack a signature.

The ability to not only detect malware but also classify which malware family it belongs to provides greater detail regarding the relative threat of a given malware. A malware detection system could alert an analyst when malicious software is found. But if the context of its malware family is not discovered, then this ambiguates the threat the malware poses. For example, if malware is delivered to a user's email inbox as an attachment and that malicious attachment is downloaded by the user, an alert may be generated to inform the cybersecurity team that a user has downloaded a malicious file. If no further information is given, such as what malware signature or malware family the software belongs to, the ability to make an informed response during triage will be difficult when similar incidents are occurring in tandem. This situation allows for a higher threat evaluation for a security incident involving relatively benign adware as opposed to something much more severe such as ransomware if the context of the threat posed by the malware is not taken into consideration.

Previous research has shown that machine learning can be useful for not only detecting malware, but also for fingerprinting malware families by learning observed malware behavior as represented by frequencies of Windows system API function calls [96] [97]. While highly accurate, this method requires the observation of every behavior a certain malware makes and is, therefore, most useful for analysis after an infected system has been remediated. To accommodate real-time detection, sequences of API calls should be observed by a machine learning model as they are executed. However, malware data tends to change in distribution as well as family type due to malware evolution and changes in operating system and commercial software exploitation techniques.

A machine learning model trained for a particular malware dataset may not provide the best results when considering dynamic data that varies with time. Therefore, machine learning models need to be updated or retrained to accommodate these changes in malware threats. Traditional machine learning methods are also time consuming, making malware family classification computationally inefficient. Therefore, in this work we have investigated the use of Extreme Learning Machine (ELM) and Online Sequential Extreme Learning Machine (OS-ELM) methods for malware signature detection. The advantage with ELM and OS-ELM is that they do not rely on computationally complex optimization algorithms like gradient descent for data training. Thus, ELM and OS-ELM can be retrained in real-time.

7.2 System Software Architecture

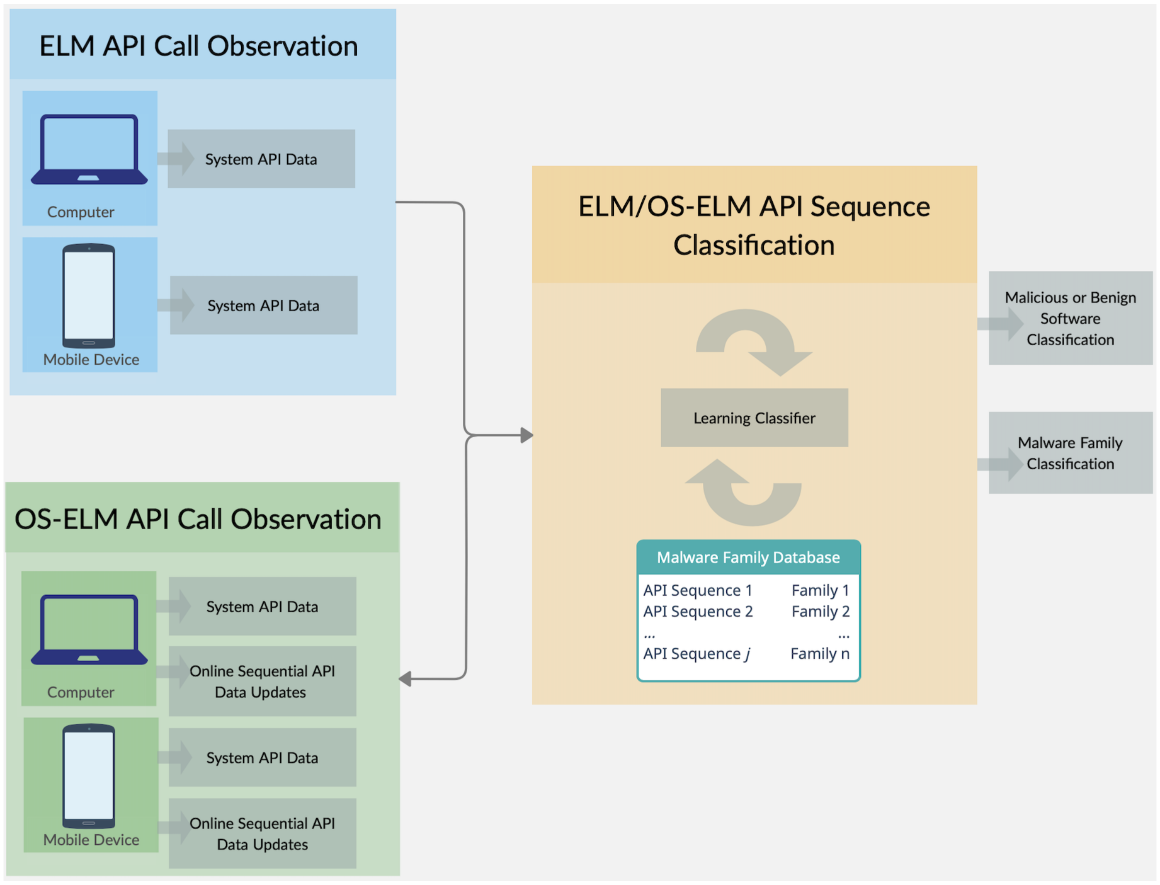


FIGURE 7.1: System Model

The proposed system model for malware family classification based upon API sequences is depicted in Fig.7.1. The methodology for observing operating system API calls consists of a running process on the system which reports sequences of such calls to the machine learning classifier. Fig.7.1 shows the use of either ELM or OS-ELM in this process, where OS-ELM would be preferred for online learning.

As we show in Section VII, ELM provides the ability to very quickly and accurately classify malware and malware family given a set of observed API call sequences. This is beneficial for when analysis of a suspicious program is required as part of a cyber-security investigation, especially when there is belief that the potentially malicious program serves as an active threat. In this case, a snapshot of observed API call sequences may be recorded and sent to the learning classifier on a remote system which is not affected by the suspicious software.

However, this reactive approach with ELM may not always be ideal. Therefore, we also propose the use of OS-ELM, which due to its self-updating nature provides a proactive approach to malware detection as well as classification. This online methodology allows for the continuous monitoring for new threats as sequences which occur after the initialization of the learning classifier are factored into the analysis of programs. Instead of only classifying programs as malicious or benign and determining malware family in an ad hoc fashion as part of incident analysis, the OS-ELM approach can be used to automatically monitor new process API calls to discover malicious software before damage is done.

7.3 Problem Statement

Consider an API call api_i at the time i. The time i is not discrete but continuous. Thus if api_i, api_{i+1}, and api_{i+2} are three sequential calls then the time difference between $i+1$ and $i+2$ is not necessarily equal to the time difference between $i+1$ and i, but rather that API calls occur in a sequential manner. The variable S_j denotes the malware signature where $j \in 1, 2, ...|S|$, where $|S|$ represents the total number of possible malware signatures. The sequence of API calls where the sequence starts at time i is denoted as the set $Seq_i = \{api_i, api_{i+1}, ...api_l\}$, where l is the total number of sequences in l. The problem statement is given below.

1. Can we predict S_j given Seq_i as the input? This involves using all values in Seq_i to predict S_i.
2. Can we predict S_j given a subset of Seq_i? This involves using only the first k api call sequences in Seq_i, where k is a subset of Seq_i. An Additional sub-problem involves determining the minimum value of k.

The difficulty of these problems lies in the large amount of API calls performed by malware. For example, we observed one variant of the $VBInject.AIE!bit$ malware performing 9,654 API calls over 1 minute and 58 seconds, where 392 API calls were made within a window of 641 nanoseconds. Manual analysis of these API calls by a human analyst is not feasible due to the wealth of actions performed. Furthermore, malware makes use of the same library of Windows API functions as benign software. Discrimination between malicious and benign behavior through API call analysis alone requires a wealth of prior knowledge of malicious API call sequences which is not possible for a human to perform. Current methods using traditional machine learning techniques are not online in nature and therefore are only capable of identifying malware according to API call sequences which were known to be malicious at the time of training. An additional difficulty concerning malware family fingerprinting is presented in terms of real-time detection and response. Any system tasked with identifying malware threats in real-time must have the ability to perform quickly and with high accuracy.

Existing methods that use Machine Learning (SVM, random forest, etc.) or Deep Learning methods (for eg. using MLP, CNN, etc.) require a significant amount of time to train. These methods are suitable when a model only needs to be trained once. However, the distribution and type of malwares tends to vary with time. Thus, a machine learning model that is trained at some time t may not be efficiently used at time $t+T$ where T is significantly larger. Therefore, we need to continuously update a model by retraining after certain intervals of time. With traditional methods this is infeasible because they take a significant amount of time to train along with increased computational costs. Therefore, online training is required to update the model with time.

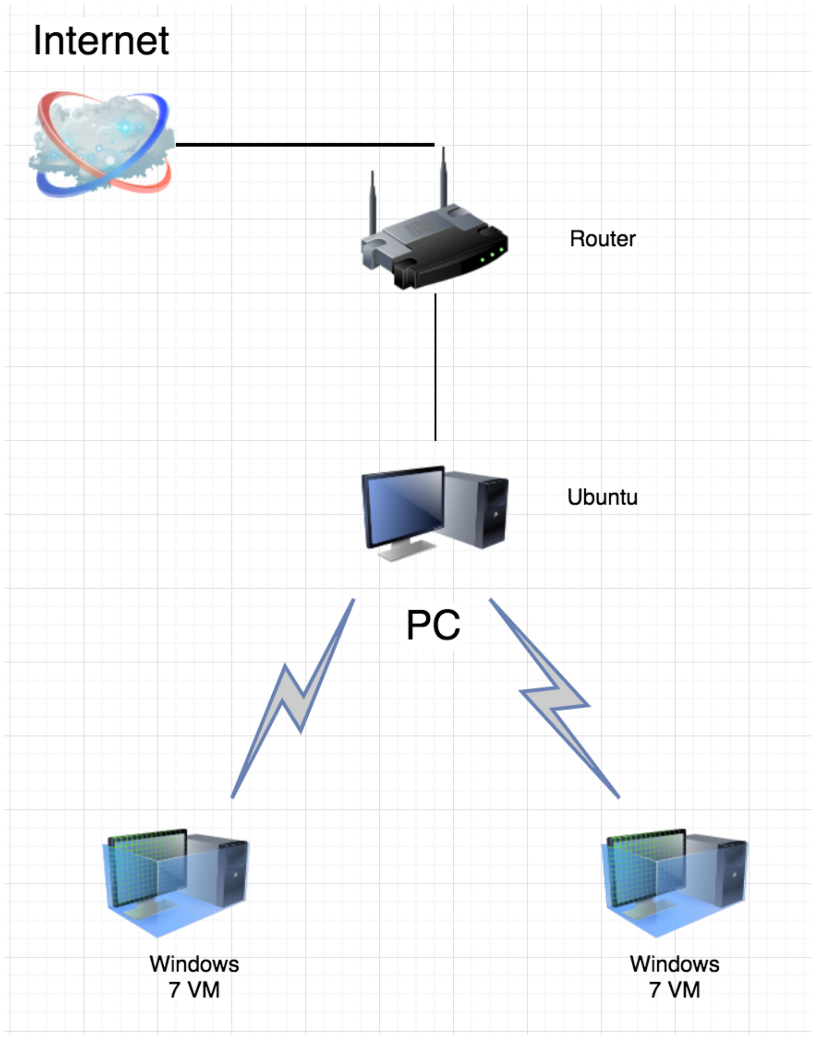


FIGURE 7.2: Cuckoo Setup on Ubuntu Host

7.4 Dynamic API call sequence classification

7.4.1 API Collection Methodology

To programmatically observe the behavior which malware performs on an infected system, we designed an environment to allow for the installation of the Cuckoo malware sandbox[22][98] for the analysis of known malware samples in a virtual Windows machine. Cuckoo is open-source software which may be installed on a host computer system for the purpose of analyzing malicious software behavior when executed within virtual systems. Figure 7.2 illustrates our setup, where Cuckoo software installed on an Ubuntu host serves malware to Windows virtual machines via a hypervisor.

Cuckoo was used in this manner to perform dynamic analysis upon 60,046 malware files obtained from VirusShare[83], an online repository for malicious software. Once a malware file is analyzed by Cuckoo, a report is generated including a list of each behavior exhibited by the malware as it was executed in a controlled environment. The process for Cuckoo report generation is described in Figure 7.3, which begins with the submission of a malicious program to Cuckoo for analysis and is followed by execution of the malware on a Windows virtual host. Actions performed by the malware are then recorded, inspected, and finally delivered in the form of a report in JSON format, which can easily be parsed for the relevant behavior artifacts.

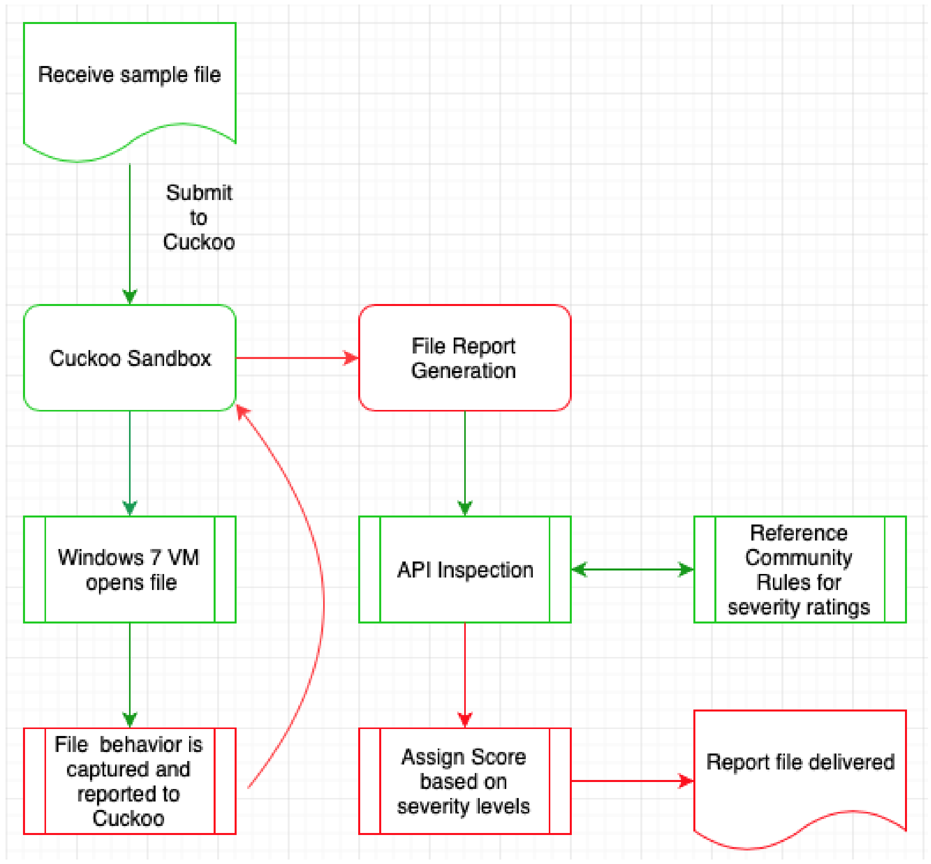
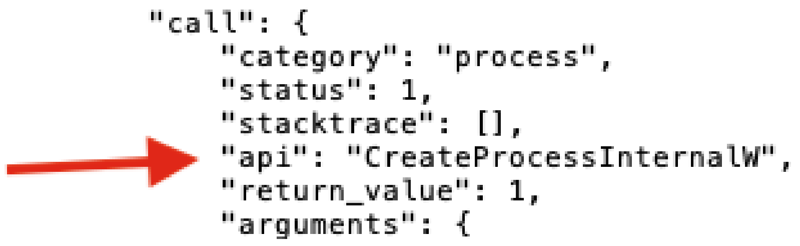

Receive sample file
Submit to Cuckoo
Cuckoo Sandbox
File Report Generation
Windows 7 VM opens file
API Inspection
Reference Community Rules for severity ratings
File behavior is captured and reported to Cuckoo
Assign Score based on severity levels
Report file delivered

Figure 7.3: Cuckoo Malware Report Generation Process

These artifacts are expressed through the Windows API system function calls made by the malware during execution. To make use of this information, we gathered the names of the API function calls as well as their timestamps to produce API call frequency reports for each malware analyzed. An example of such an API call can be seen in Figure 7.4, where the Cuckoo report shows the Windows API function “CreateProcessInternalW” used for malicious code injection.



```
"call": {
    "category": "process",
    "status": 1,
    "stacktrace": [],
    "api": "CreateProcessInternalW",
    "return_value": 1,
    "arguments": {
```

FIGURE 7.4: Example of API Used for Malicious Code Injection

Of the total number of malware files analyzed, 29,868 files were unique according to their file hashes. VirusTotal[56] is an online resource for comparing software to known malware signatures as a means of malware detection. This service was used to compare the hash of each unique malware file to malware identified by various antivirus (AV) vendors. Each of the AV vendors who reported that a given malware file was indeed malicious assigned a name for the family to which the malware belonged. These malware family names vary as there is no set standard nomenclature. We chose to follow the naming convention used by Microsoft AV. Since a version of their AV software is available on all modern Windows operating systems, this seems to be an appropriate baseline. These Microsoft AV signatures were referenced to assign a malware family to each malware file, resulting in 318 family designations for the full set of unique malwares. Of these family designations, 138 corresponded to a single malware file each, while a total of 247 malware family designations mapped to less than 10 malware files each.

7.4.2 API Sequence Dataset

The API sequence data for each malware family was combined for each malware family designation to represent known API sequences for a given malware family. We found that sequences of 10 API call names produced the best accuracy in our experimentation, as this number was large enough to discriminate against similar API call sequences for different malware families which could interfere with learning. For example, during training we found that smaller sequences of 2-5 API calls (such as {LdrLoadDll, LdrLoadDll, LdrLoadDll} would often occur amongst many malware family labels. However, we found that such sets of 10 API calls rarely were duplicated amongst different labels. These malware family sequence files were then used for training and validation of the machine learning models used.

7.5 Analysis of ELM and OS-ELM

7.5.1 Extreme Learning Machine

```
Algorithm 1: Extreme Learning Machine (ELM)
Input: $\boldsymbol{Seq} = \{Seq_j\}$ , $\boldsymbol{T} = \{S_j\}$ , $j = 1, 2, \ldots, N$
Output: class label $\boldsymbol{S_j}$
1   Initialization: Randomly assign weight matrix $w$ and the bias vector $b$.
      • Calculate matrix $\boldsymbol{H}$ based on Eq. 7.3
      • Calculate parameter vector $\beta$ based on Eq. 7.6
      • Obtain final output $\boldsymbol{S_j}$ based upon Eq. 7.1
```

The Extreme Learning Machine (ELM) algorithm makes use of single hidden layer feedforward neural networks (SLFNs) along with random input weights, which performs much faster than the traditional back-propagation algorithm [99][100][101]. A SLFN network includes three layers: an input layer, a single hidden layer, and an output layer as described in Fig. 7.5. Such a network is useful in classification problems, especially in complex problems such as sequence classification due to its time efficiency when compared to traditional training algorithms such as LSTM. In our problem, the input is a set of API call sequences Seq_i which is composed of $\{api_1$, api_2, api_3, ..., $api_l\}$. The output of the network with L hidden nodes can be represented by Eq. 7.1, with a set of N training samples with input as $(Seq_1, Seq_2, ...Seq_N)$ and output as $(S_1, S_2....., S_N)$.

$$S_j = \sum_{i=1}^{L} \beta_i g\left(a_i \cdot Seq_j + b_i\right) \quad j = 1, 2, \ldots, N \tag{7.1}$$

Here, the output weight is expressed as $\beta_i \epsilon \mathbb{R}^m$, the input weight and bias values are expressed with $a_i \epsilon \mathbb{R}^n$ and $b_i \epsilon \mathbb{R}$. Assuming equation 7.1 has zero error, it can be expressed compactly in matrix format as

$$\boldsymbol{H\beta} = \boldsymbol{T} \tag{7.2}$$

where

$$\boldsymbol{H} = \begin{bmatrix} g\left(a_1 \cdot Seq_1 + b_1\right) & \dots & g\left(a_L \cdot Seq_1 + b_L\right) \\ \vdots & \ddots & \vdots \\ g\left(a_1 \cdot Seq_N + b_1\right) & \dots & g\left(a_L \cdot Seq_N + b_L\right) \end{bmatrix}_{N\times L} \tag{7.3}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L\times m} = \begin{bmatrix} \beta_{11} & \dots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{N1} & \dots & \beta_{Lm} \end{bmatrix}_{L\times m} \tag{7.4}$$

$$\boldsymbol{T} = \begin{bmatrix} S_1 \\ \vdots \\ S_2 \end{bmatrix}_{N\times m} \tag{7.5}$$

$\boldsymbol{T}$ is the training data target matrix and $\boldsymbol{H}$ is referred to as the hidden layer output matrix of the ELM[99], where a_i and b_i are not tuned during training, but are randomly assigned[102]. Accordingly, Eq. 7.2 becomes a linear system with output weights $\boldsymbol{\beta}$ estimated as

$$\boldsymbol{\beta} = \boldsymbol{H}^{\dagger}\boldsymbol{T} \tag{7.6}$$

where $\boldsymbol{H}^{\dagger}$ is the Moore-Penrose pseudo-inverse[103] of the hidden layer output matrix **H**.

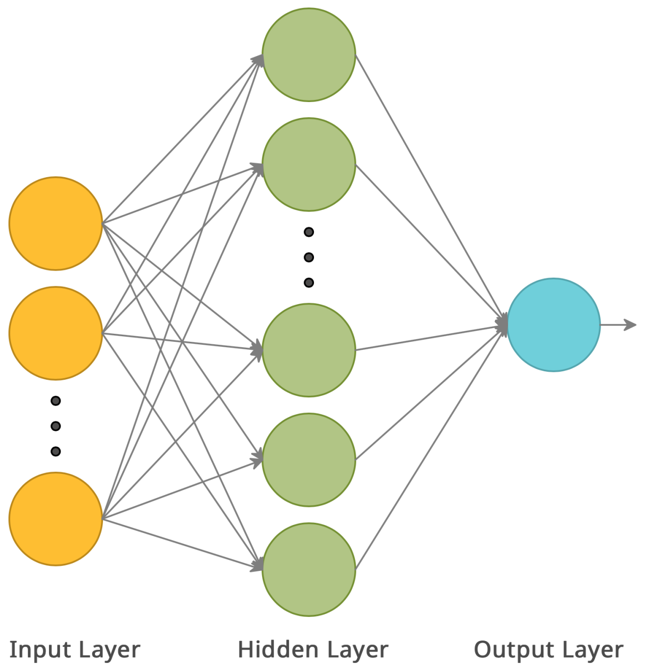


FIGURE 7.5: An ELM Network

7.5.2 Online Sequential Extreme Learning Machine

ELM assumes that all the training data is available up front and in a single batch. Liang et al.[102] described how ELM could be extended to perform training when data arrives sequentially through the Online Sequential Extreme Learning Machine (OS-ELM) algorithm. This allows for continued training when additional input data is provided after the initialization of computation, allowing for an online approach which has the potential to be of greater value to real-time systems. The OS-ELM algorithm consists of two phases: 1) the initialization phase and 2) the sequential learning phase.

7.5.2.1 Initialization Phase

In this phase, the initial hidden layer output matrix needs to be defined based on the initial chunk of training data. Let us consider a small chunk D_0 of the training dataset D

$$D_0 = \{(Seq_i, t_i)\}_{i=1}^{N_0} \tag{7.7}$$

where

$$D = \{(Seq_i, t_i)\,| x_i \,\epsilon\, R^n, \;\; t_i \,\epsilon\, R^m, \;\; i = 1, \ldots, N_0\} \tag{7.8}$$

From here, the random input weights and bias, a_i and bi respectively, are assigned and the initial hidden layer output matrix $\boldsymbol{H}_0$ is calculated as seen in Eq. 7.9.

$$\boldsymbol{H}_0 = \begin{bmatrix} g\left(a_1 \cdot Seq_1 + b_1\right) & \ldots & g\left(a_L \cdot Seq_1 + b_L\right) \\ \vdots & \ddots & \vdots \\ g\left(a_1 \cdot Seq_{N_0} + b_1\right) & \ldots & g\left(a_L \cdot Seq_{N_0} + b_L\right) \end{bmatrix}_{N \times L} \tag{7.9}$$

The initial output weight $\boldsymbol{\beta}^0$ is computed as follows:

$$\boldsymbol{\beta}^0 = \boldsymbol{P}_0 \boldsymbol{H}_0^T \boldsymbol{T}_0 \tag{7.10}$$

where

$$\boldsymbol{P}_0 = \left(\boldsymbol{H}_0^T \boldsymbol{H}_0\right)^{-1} \tag{7.11}$$

and

$$\boldsymbol{T}_0 = \left[\boldsymbol{t}_1, \ldots, \boldsymbol{t}_{N_0}\right]^T \tag{7.12}$$

7.5.2.2 Sequential Phase

With the previous phase calculations performed, OS-ELM provides the ability to consider observations on training data beyond the initial set of X_0. Eq. 7.13 describes how the $(k+1)$th chunk of new observations is presented, where $k = 0$ and N_{k+1} denotes the number of observations in the $(k+1)$th chunk.

$$D_{k+1} = \{(Seq_i, t_i)\}_{i=\left(\sum_{j=0}^{k} N_j\right)+1}^{\sum_{j=0}^{k+1} N_j} \tag{7.13}$$

The partial hidden layer output matrix is calculated, followed by a calculation of the output weight $\boldsymbol{\beta}^{k+1}$

$$\boldsymbol{P}_{k+1} = \boldsymbol{P}_k - \boldsymbol{P}_k \boldsymbol{H}_{k+1}^T \left(\boldsymbol{I} + \boldsymbol{H}_{k+1} \boldsymbol{P}_k \boldsymbol{H}_{k+1}^T\right)^{-1} \boldsymbol{H}_{k+1} \boldsymbol{P}_k \tag{7.14}$$

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^{k} + \boldsymbol{P}_{k+1}\boldsymbol{H}_{k+1}^{T}\left(\boldsymbol{T}_{k+1} - \boldsymbol{H}_{k+1}\boldsymbol{\beta}^{k}\right) \tag{7.15}$$

Equations 7.14 and 7.15 show how the output weights are recursively updated based on the results of the last iteration as well as the present data in the current iteration. The value of k is increased by one and the Sequential Phase is repeated. This cycle will continue until the maximum value of k is reached and the final output weights determine the value of S_j in an online manner

7.6 Experimental Results

Here we present the results of our experimentation with machine learning models for sequence classification. Our experiments were composed of three test groups, each made up of three malware family classes as described in Table 7.1. These malware families were chosen randomly from our malware dataset where the total number of API function calls were similar. This allowed us to experiment with a robust number of API calls while maintaining balanced datasets.

7.6.1 Question 1

In this section we address the first research question:

TABLE 7.1: Malware Test Groups

Test Group	Label	Malware Families
Test Group 1	0	Trojan:JS/Redirector.ARA!MTB
	1	TrojanClicker:JS/Faceliker.C
	2	Exploit:HTML/IframeRef
Test Group 2	0	TrojanClicker:JS/Faceliker.I
	1	TrojanClicker:JS/Faceliker.C
	2	PUA:Win32/Presenoker
Test Group 3	0	PUA:Win32/DownloadGuide
	1	SoftwareBundler:Win32/Prepscram
	2	PUA:Win32/Presenoker

- Can we predict S_j given Seq_i as the input? This involves using all values in Seq_i to predict S_i.

7.6.1.1 ELM Analysis

Our ELM implementation involved the use of 1000 hidden units with a 60/40 split between training and test sets. This resulted in overall accuracies of 59% for Test Group 1, 62% for Test Group 2, and 91% for Test Group 3. Precision, recall, and F1 scores for these groups are presented in Tables 7.2, 7.3, and 7.4, respectively. In addition to accuracy, we found it relevant to track the amount of time taken for each ELM model to execute and make predictions. ELM Test 1 completed in 1 second, ELM Test 2 completed in 2 seconds, while ELM Test 3 completed in 3 seconds, as reflected in Figure 7.7.

Table 7.2: ELM Statistics for Test Group 1

	Precision	Recall	F1-score	Support
0	0.50	0.37	0.43	12287
1	0.70	0.97	0.81	12439
2	0.49	0.43	0.46	12112
Accuracy			0.59	36838
Macro Avg.	0.56	0.59	0.57	36838
Weighted Avg.	0.57	0.59	0.57	36838

Table 7.3: ELM Statistics for Test Group 2

	Precision	Recall	F1-score	Support
0	0.49	0.79	0.61	12150
1	0.48	0.20	0.29	12274
2	0.89	0.88	0.88	11944
Accuracy			0.62	36368
Macro Avg.	0.62	0.62	0.59	36368
Weighted Avg.	0.62	0.62	0.59	36368

Table 7.4: ELM Statistics for Test Group 3

	Precision	Recall	F1-score	Support
0	0.91	0.86	0.89	13902
1	0.88	0.96	0.92	14104
2	0.94	0.90	0.92	13918
Accuracy			0.91	41924
Macro Avg.	0.91	0.91	0.91	41924
Weighted Avg.	0.91	0.91	0.91	41924

7.6.1.2 OS-ELM Analysis

Our OS-ELM implementation involved a 60/40 split between training and test sets and performed similarly to ELM in terms of overall accuracy and execution time. However, we observed that the accuracy of the OS-ELM models varied in relation to the initial training size. Figure 7.6 describes the different accuracies resulting from different initial training sizes for Test Groups 1-3.

We observe that for our datasets, lower initial training sizes (less than approximately 50,000 - 60,000 training samples) result in accuracies which fluctuate significantly and are much less than the accuracy values expressed through ELM. However, larger initial training sizes result in the same accuracies as the ELM models, though the exact value of the initial training sizes necessary to reach this limit varies. Test Group 1 achieved an overall accuracy of 33% with an initial training size of 55,000 samples, yet when we add just 1,000 more samples to the initial training size we observe an accuracy of 59%. Similarly, Test Group 2 achieved a 34% accuracy with an initial training size of 54,000 samples, while an increase to 55,000 results in an accuracy of 62%. Finally, Test Group 3 with an initial training size of 62,000 samples resulted in an accuracy of 45% and then an accuracy of 91% with 63,000 samples in the initial training size.

We also experimented with sequential batch sizes of 64, 250, 500, and 1,000 samples. We observed that smaller batch sizes resulted in higher accuracies prior to hitting the accuracy limits described above, but there was no effect of different batch sizes afterward.

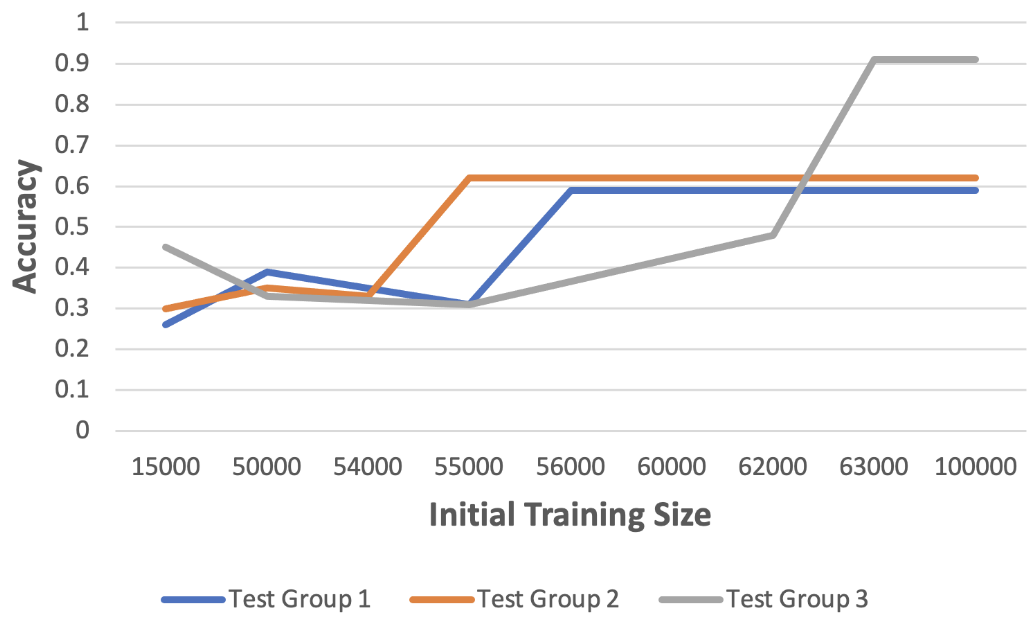


FIGURE 7.6: OS-ELM Accuracy by Initial Training Size for Test Groups 1 - 3

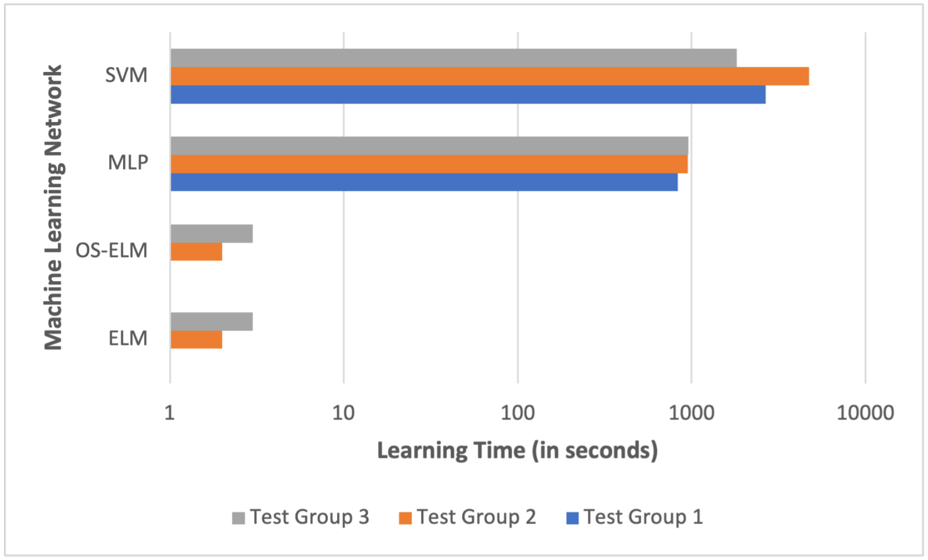


FIGURE 7.7: Learning Time for SVM, MLP, OS-ELM, and ELM

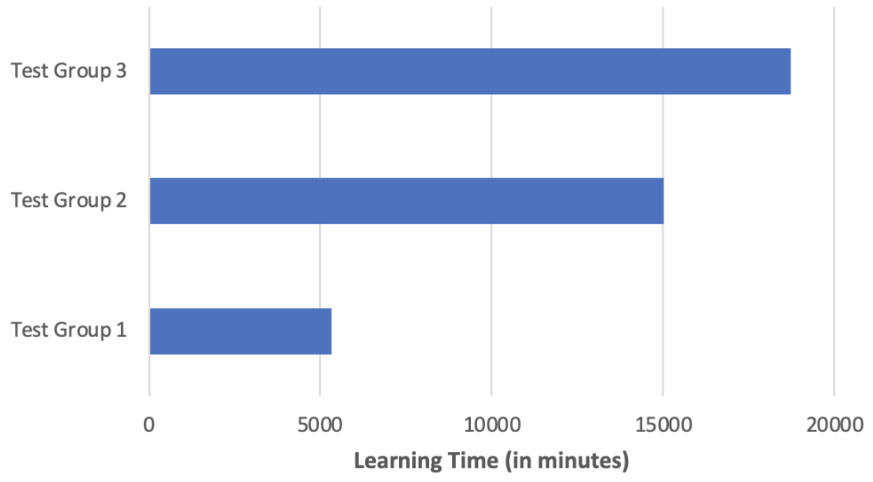


Figure 7.8: Learning Time for Test Groups using LSTM

7.6.1.3 LSTM

In our LSTM methodology, we implemented a five-layer sequential model including embedding, dropout, LSTM, dropout, and dense layers. The LSTM layer consisted of 100 neurons and dropout was set to 20%. Our model was compiled using binary cross-entropy for the loss function and was optimized using the Adam algorithm. These networks were trained with a 60/40 split between training and test sets.

We found that on average, LSTM performed with a lower overall accuracy as ELM and OS-ELM and with much more time taken to train and test the models. The overall accuracy for Test Group 1 was 63% with a running time of 3 days, 16 hours, 39 minutes. The accuracy for Test Group 2 was 60% with a running time of 10 days, 10 hours, 25 minutes. Finally, the accuracy for Test Group 3 was 80% with a running time of 13 days, 5 minutes, as reflected in Figure 7.8. In contrast, ELM learning for these test groups completed within 3 seconds.

7.6.1.4 MLP and SVM

In addition to ELM and LSTM, we considered the use of a multi-layer perceptron (MLP) model as well as Support Vector Machines (SVM) for malware family predictions. These models were trained with a 60/40 split between training and test sets.

MLP Test 1 completed in 13 minutes, 52 seconds. MLP Test 2 completed in 15 minutes, 54 seconds. MLP Test 3 completed in 19 minutes.

Table 7.5: MLP Statistics for Test Group 1

	Precision	Recall	F1-score	Support
0	0.51	0.44	0.47	12287
1	0.79	0.99	0.88	12439
2	0.50	0.44	0.47	12112
Accuracy			0.63	36838
Macro Avg.	0.60	0.62	0.61	36838
Weighted Avg.	0.60	0.63	0.61	36838

Table 7.6: MLP Statistics for Test Group 2

	Precision	Recall	F1-score	Support
0	0.50	0.95	0.66	12150
1	0.60	0.08	0.15	12274
2	0.94	0.93	0.93	11944
Accuracy			0.65	36368
Macro Avg.	0.68	0.65	0.58	36368
Weighted Avg.	0.68	0.65	0.57	36368

SVM Test 1 completed in 44 minutes, 25 seconds. SVM Test 2 completed in 1 hour, 18 minutes, 47 seconds. SVM Test 3 completed in 30 minutes, 21 seconds.

The overall accuracies for the MLP approach were slightly higher than with ELM and OS-ELM, while the SVM approach performed poorly on average in comparison

Table 7.7: MLP Statistics for Test Group 3

	Precision	Recall	F1-score	Support
0	0.95	0.91	0.93	13902
1	0.92	0.98	0.95	14104
2	0.97	0.94	0.95	13918
Accuracy			0.94	41924
Macro Avg.	0.94	0.94	0.94	41924
Weighted Avg.	0.94	0.94	0.94	41924

as shown in Tables 7.5 - 7.10. Despite the best accuracy achieved by the MLP, we can see that it was tantamount to the performance of the ELM. In addition, the accuracies using ELM were achieved with a mean learning time of 2 seconds, compared to that of 16 minutes for the MLP. This makes ELMs an attractive option for performing real-time malware analysis on computing environments.

Table 7.8: SVM Statistics for Test Group 1

	Precision	Recall	F1-score	Support
0	0.36	0.36	0.36	12287
1	0.55	0.25	0.34	12439
2	0.35	0.55	0.43	12112
Accuracy			0.38	36838
Macro Avg.	0.42	0.39	0.38	36838
Weighted Avg.	0.42	0.38	0.38	36838

Table 7.9: SVM Statistics for Test Group 2

	Precision	Recall	F1-score	Support
0	0.42	0.92	0.58	12150
1	0.44	0.07	0.12	12274
2	0.87	0.58	0.69	11944
Accuracy			0.52	36368
Macro Avg.	0.58	0.52	0.46	36368
Weighted Avg.	0.57	0.52	0.46	36368

Table 7.10: SVM Statistics for Test Group 3

	Precision	Recall	F1-score	Support
0	0.92	0.88	0.90	13902
1	0.90	0.97	0.93	14104
2	0.95	0.92	0.93	13918
Accuracy			0.92	41924
Macro Avg.	0.92	0.92	0.92	41924
Weighted Avg.	0.92	0.92	0.92	41924

7.6.2 Question 2

In this section we address the second research question:

- Can we predict S_j given a subset of Seq_i? This involves using only the first k api call sequences in Seq_i, where k is a subset of Seq_i. An Additional sub-problem involves determining the minimum value of k.

Our experiments show that a significantly reduced subset of sequential API calls made by a given malware can be used to fingerprint that malware as a member of a certain family. Table 7.11 describes the variation of accuracy when reducing the number of API calls is reduced for each label (malware family) in the test group. For example, Test Group 3, which has an accuracy of 91% when using its full dataset, maintains an accuracy of 91% when reduced to as little as 6.25% of its total dataset before reducing in accuracy to 39% when only 3.125% of the dataset is used.

We find similar accuracies for Test Group 2 and 3, which both maintain accuracies like that resulting from the full dataset when reduced to just 12.5% of their datasets. It should be noted that the reduction in datasets is relative to the beginning of the malware sequences, inferring that malware families can be identified from the first few API call sequences.

Table 7.11: Changes in ELM Accuracy With Reduced Datasets

Test Group	% of API Sequences	Accuracy
Test Group 1	100% (30698)	59%
	50% (15349)	60%
	25% (7674)	59%
	12.5% (3837)	51%
	6.25% (1918)	34%
	3.125% (959)	34%
Test Group 2	100% (30306)	62%
	50% (15153)	62%
	25% (7576)	61%
	12.5% (3788)	54%
	6.25% (1894)	36%
	3.125% (947)	34%
Test Group 3	100% (34936)	91%
	50% (17468)	90%
	25% (8734)	91%
	12.5% (4367)	91%
	6.25% (2183)	91%
	3.125% (1091)	39%

This experiment has the additional outcome of showing that our malware family fingerprinting is resistant to obfuscation of malware behavior. As a relatively small subset of API call sequences is all that is needed to classify malware, attempts to resist fingerprinting can be defeated by making several classification attempts against a single malware.

For example, Table 7.11 shows that the minimum value of k which produced 91% accuracy for Test Group 3 was 2,183 sequences, or just 6.25% of the total number of

API sequences. Assume that a new malware belonging to one of the same families from this test group has been programmed to execute API sequences in a different order. Then it would be possible to randomly select k API sequences from the total set of sequences observed from the new malware's behavior to produce an accurate classification.

7.7 Discussion and Recommendations

Previous research has shown that computationally expensive machine learning algorithms such as LSTM can be used for discerning malware from benign software. However, our research shows that ELM can also be used to accurately predict malware family classification through analysis of Windows API call sequences with the benefit of significantly reduced prediction time. Additionally, a relatively small number of API calls in a sequence can be used for this detection.

This methodology can be extended to serve as a real-time malware detection system, where API call sequences may be monitored on a computer and compared against known malware sequences. The usage of ELM for malware sequence classification provides this ability in as little as a second in our experimentation, which is far superior in training speed to LSTM, MLP, or SVM. The accuracy of ELM is slightly reduced in comparison to MLP. However, we argue that the benefit of ELM's much faster learning time is significant enough to allow real-time detection. Additionally, the online nature of OS-ELM allows for the speed of ELM while providing the opportunity for greater accuracy on unknown malware due to the real-time update of the machine learning model.

Chapter 8

Bespoke Automated Malware Risk Classification

In the previous chapters we described how machine learning can aid malware analysis. In this chapter, we show how these methodologies can be combined into a framework for automated malware threat determination. Several existing scoring systems are presented and are shown to lack the ability to take into account the differences in security posture each organization has and how that relates to malware threat analysis.

8.1 Introduction

Organizations which have a mature cybersecurity program are well-versed in the language of risk. Regulatory and standards compliance is a chief concern amongst enterprises which handle protected health information, credit card transaction data, or student information. The challenges of protecting the confidentiality, integrity, and availability of such information for authorized use requires regular and thorough assessment of organizational policies and procedures. This promotes a healthy security posture and therefore greater confidence in the efficacy of a cybersecurity program.

However, the process of establishing risk ratings and generating appropriate response procedures is not necessarily straight-forward. The diversity of industry results in organizations which do not face the same risks, or even value the same risks equally. When we define a threat as an agent or event which has the potential to do harm and define a risk as the potential for loss or damage should the threat occur, we can visualize how not all threats and risks are equal for all organizations. A banking institution will be concerned with preventing a breach of customer account information, while a library may not face the same risk. A university with a medical school is obligated to protect student records, patient health information, and payment transactions for tuition and fees. A small business in the dining industry will be concerned with secure payments but may not otherwise transmit or store any customer information. In each of these cases the same threats may exist, but each organization's response to these threats will be different based upon the potential risk impact. Determining risk impact is not trivial because it requires a deep understanding not only of the industry within which an organization operates, but also the various internal environmental factors involved in such an organization. Because of this, methods of ascertaining risk are often dependent upon the value of a related threat or vulnerability.

There are several frameworks currently available which can be used to identify and qualify organizational risk, such as CMMI[104], NIST 800-53[105], and ISO 27001[106]. These frameworks identify security and privacy controls to minimize information security risks. Risk rating systems such as OCTAVE Allegro[61], Common Vulnerability Scoring System (CVSS)[2], and Infocon[107] allow for detailed organizational risk assessments for software vulnerabilities and internet service disruptions. However, these frameworks fail to provide a robust methodology for assessing and responding to cybersecurity risks directly associated with malware.

Malware, i.e., malicious software, presents a threat to computing environments in the office, at home, and in travel. Malware developers create software to exploit the flaws in any platform and application which suffers a vulnerability in its defenses, be it through unpatched known attack vectors or zero-day attacks for which there is no current solution. Malware which successfully exploits such a vulnerability produces an information security incident. Security incidents may involve the loss of functionality of a system, compromise of user account credentials, unauthorized access into a system, or any number of conditions which compromise the confidentiality, integrity, or availability of a system or application.

While all successful malware compromises require a measure of response in order to restore faith in the proper working order of a system, not all malware attacks are created equal. For large organizations, the number of live information security incidents can be staggering. Prioritization of incidents based upon levels of severity is necessary for the quick elimination of the most severe threats and the continued monitoring and assessment of threats not yet handled. This is especially true for organizations with a relatively small information security incident response team. In situations where high volumes of security incidents may be present, an automated means of prioritization is essential to help incident analysts to quickly triage and respond in an appropriate manner.

Cybersecurity vendors such as WatchGuard provide automated tools which assess the criticality of malware[108], however their rubric by which a severity score is measured is designed to match the needs of their broad range of customers. This results in a "one size fits all" approach to malware severity scoring which does not accurately reflect the criticality of a certain malware infection with regard to the mitigating security strategies of a given organization. Therefore, an organization which has fewer security strategies may be devastated by a malware outbreak which would be simple for another organization to remediate. The current methods for evaluating malware severity are therefore misleading and may result in greater risk as a result of misguided confidence in a low severity score provided by these means. In this chapter we present a bespoke automated methodology for evaluating malware severity and show how such a strategy is more accurate and beneficial to an organization than generalized automated solutions.

8.2 Analysis of Existing Risk Assessment Methodologies

8.2.1 CVSS

The Common Vulnerability Scoring System (CVSS) is an open framework for communicating the characteristics and severity of software vulnerabilities[2]. It was created by the Forum of Incident Response and Security Teams (FIRST), in coordination with the National Institute of Standards and Technology (NIST). The CVSS scoring system was adopted as part of the Payment Card Industry Data Security Standard (PCI DSS)[2] and as a result has been used in assessments of software security.

The CVSS is designed to measure the severity of a vulnerability, not risk. This means that CVSS scores refer to the potential criticality of a software vulnerability in any operating environment and does not take into account any mitigating factors which might impact the risk associated with the vulnerability, such as defense-in-depth strategies, data backups, or security technologies which limit access to a vulnerable system, to name a few.

NIST provides an online calculator for CVSS severity scores[109] which allows the user to toggle various score impact factors. Figure 8.1 illustrates the result of one such calculation, where choices were made so that the various metrics might reflect a ransomware infection. The metrics involved include exploitability and impact factors, ranging from the complexity required from an attack in order to successfully exploit a vulnerability to the affect of the attack on the confidentiality, integrity, and availability of data. As a result, we received an overall score of 7.8 out of 10.

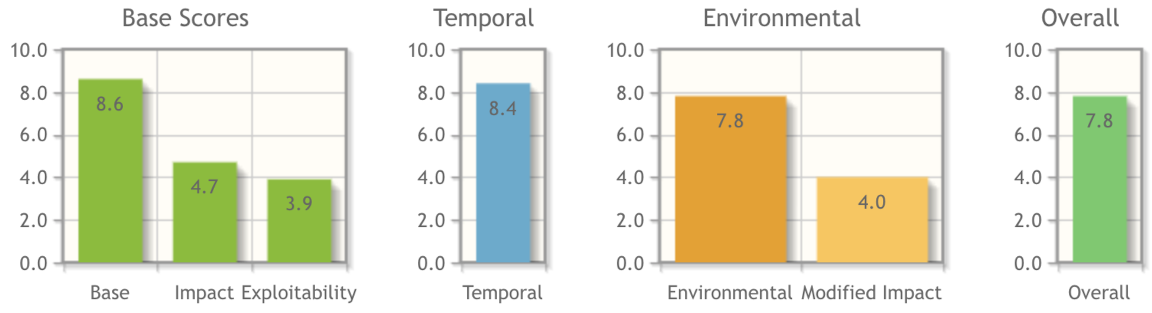


Figure 8.1: Example CVSS Severity Scores

The difficulty of applying CVSS to malware lies in the lack of consideration for mitigating security controls. As CVSS is designed to evaluate software vulnerabilities, there is a limited range of possible values for the considered metrics. From a software security perspective, severity can have a true or false value for metrics such as impact on data integrity in order to evaluate the scope of potential threat impact. However, malware and software vulnerabilities are separate in that a vulnerability is an aspect of the condition of software on a system while malware is an agent which potentially exploits a vulnerability. Thus, as in the case of our score of 7.8 for ransomware, this severity value addresses the vulnerabilities associated with a ransomware attack yet does not reflect how the presence of antivirus software, data backups, and disaster recovery methods might impact the true severity of a ransomware attack. This results in a severity value which only considers the vulnerability and not the threat which may exploit it.

8.2.2 Infocon

Infocon is a risk rating scale created by the SANS Internet Storm Center (ISC) as means of reflecting changes in malicious internet traffic and the possibility of disrupted connectivity[107]. The focus of this risk assessment methodology is on the impact which certain vulnerabilities and malware, especially viruses and worms, actively have on the Internet infrastructure. Of particular interest to the current research is the risk rating rubric employed by Infocon, which is as follows:

- +2 Slammer-like impact on Internet wide operations
- +2 Remote arbitrary code execution
- +2 No vendor patch or effective mitigation is available
- +2 Active exploitation of vulnerability
- +1 Affects current version of up to date software
- +1 Affects widely deployed software
- +1 Relatively easy to exploit
- +1 Proof of concept code is available

- +1 Affects current version of up to date software
- +1 Affects a Microsoft OS or Adobe application
- +1 Wormable
- -1 Affects obscure or obsolete OS or application
- -1 Requires user intervention to run
- -1 IDS/IPS rules or other detective controls are available
- -1 Major anti-virus vendors can detect and clean malware
- -1 Mainstream media and everyone else has already covered issue
- -1 Vendor has released an advisory/ bulletin/ announcement (and decent workaround)

The resulting score applied to a malware threat by this rubric is then compared to a certain status, as follows:

- Infocon Green (0-5)
  - Everything is normal. No significant new threat known.
- Infocon Yellow (6-9)
  - We are currently tracking a significant new threat. The impact is either unknown or expected to be minor to the infrastructure. However, local impact could be significant. Users are advised to take immediate specific action to contain the impact.

- Infocon Orange (10+)
  - A major disruption in connectivity is imminent or in progress.
- Infocon Red (Conditional)
  - Loss of connectivity across a large part of the internet.

An example usage of this rubric and resulting status can be observed from the ISC's analysis of the Wannacry/WannaCrypt ransomware outbreak in May of 2017[110]. While the exact Infocon rubric calculation was not explicitly presented, the resulting status was elevated from Green to Yellow. This means that the risk score was evaluated to be between 6 and 9. This score is likely related to the remote arbitrary code execution (+2), active exploitation (+2), and lack of a vendor patch (+2) resulting from Wannacry's exploitation of the ETERNALBLUE Windows SMB zero-day vulnerability[111].

The Infocon rubric considers malware threats, rather than vulnerabilities as seen in CVSS. This makes this system more applicable for considering the severity of a wide range of malware threats. However, the Infocon rubric only considers the actions of malware and not any mitigating security controls present within an organization targeted by the malware threat. This can be illustrated by again considering the threat of Wannacry in an organization which makes exclusive use of the Linux operating system for workstations and servers. Since Wannacry exploited a Windows SMB vulnerability, this condition for the successful ransomware attack would be unavailable and this malware would present little to no threat to such an organization. The lack of an ability to reflect these organizational factors into the Infocon risk rating limits the efficacy of this methodology beyond a generic severity score.

8.2.3 WatchGuard TDR Threat Score

The Watchguard Threat Detection and Response (TDR) is a service which classifies cybersecurity threats observed by its network security products such as the Firebox and APT Blocker sandbox[108]. These threats may be related to malware or the exploitation of security vulnerabilities, and are evaluated according to heuristic analysis, comparison of the current threats to known malicious activity according to shared threat information feeds, and sandbox analysis of malware activity for known malicious behavior on an infected system. The resulting threat information is then assigned a criticality score according to the following rubric:

- 10 (Critical)
  - Scored based on host indicator threat feed, Malware Verification Service confirmation, or both, and critical network alerts. This score can also indicate that Host Ransomware Prevention was triggered and the Host Sensor action to prevent it failed.
- 9 (Critical)
  - Scored based on the result of APT Blocker sandbox analysis.

- 8 (Severe)
  - Scored based on host Indicator threat feed, Malware Verification Service confirmation, or heuristics identification of multiple behaviors for the same object. An indicator can also be assigned this score as a result of APT Blocker sandbox analysis.
- 7 (High)
  - Scored based on network activity, heuristics identification of multiple behaviors for the same object, or third-party network activity. An indicator can also be assigned this score as a result of APT Blocker sandbox analysis.
- 6 (High)
  - Scored based on network activity or heuristics identification of multiple behaviors for the same object. A network indicator can also be assigned this score when APT Blocker on a Firebox blocks a known threat.
- 5 (Investigation)
  - Scored based on heuristics identification of multiple potential malicious process behaviors. A network indicator can also be assigned this score when APT Blocker on a Firebox blocks a known threat.
- 4 (Medium)

  - Medium priority ranked indicators, including third-party vendor scores, primarily network activity indicators. A network indicator can also be assigned this score when APT Blocker on a Firebox blocks a known threat.

- 3 (Low)
  - Low priority ranked indicators, including WatchGuard and third-party vendor scores, primarily network indicators.
- 2 (Suspect)
  - Low fidelity file heuristics without other correlation. Indicators with this score do not appear on the Indicators page.
- 1 (Remediated)
  - Identified host indicator has been remediated on the host.
- 0 (Known Good)
  - Host does not have any detected indicators or the object is on the allowlist.

An example TDR threat score of 10 might then result from a detected Wannacry ransomware infection, as a ransomware event in this rubric is assigned the highest criticality. However, this presumes that the TDR system is able to successfully detect ransomware and/or the malware in question has a known antivirus signature and/or the identifiers of the malware (such as MD5 hash) are known and included in a malware threat feed. Without these positive identifiers of the ransomware, a score of 9 is possible if the malware behavior is detected in WatchGuard's APT Blocker sandbox. Otherwise, a lower score may be assigned which may impede correct threat response prioritization. This places a high burden on Watchguard's sandbox to correctly identify the malware threat.

The use of a malware sandbox, threat intelligence feeds, and heuristics presents a more sophisticated approach to automated malware analysis when compared to a stand-alone scoring method as previously seen in CVSS and Infocon. However, processes TDR uses for these evaluations are largely unknown and there is no means for adjusting threat priorities in relation to an organization's security posture. Given the assumption that many organizations make use of the Windows operating system, a threat such as Wannacry which exploits a Windows SMB vulnerability would again present as a critical risk for any organization. This fails to capture the true risk conditions for an organization which makes exclusive use of the Linux operating system or has mitigating security controls such as daily data backups to a remote site which could be recovered in the event of a ransomware incident.

8.3 WOPR: Automated Bespoke Malware Threat Modeling

In this section we describe WOPR, our approach to bespoke malware threat evaluation which makes use of the tools and techniques described in previous chapters for malware threat classification. Figure 8.2 describes the flow of our threat modelling approach, which makes use of the following steps to automatically investigate malware threats and provide a recommended response:

1. The suspected malware executable is submitted to WOPR for automated analysis.

2. The executable is analyzed in an automated sandbox to extract operating system API function calls.

3. Machine learning is used to classify the submitted executable as malicious or benign according to API call information. If the executable is malicious then malware family classification is additionally performed using machine learning.

4. With the malware classified as a certain threat, automated analysis of the risk associated with the threat for a particular organization is performed.

5. Finally, an automated malware incident response recommendation is provided.

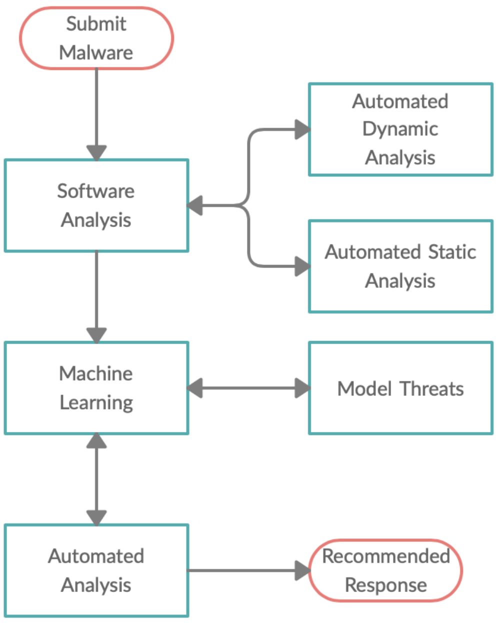


Figure 8.2: WOPR flow diagram describing threat category determination.

Steps 1 and 2 described above correspond to the use of the Cuckoo Sandbox for automated API call reporting as shown in Chapter 3. The machine learning classification tasks described in step 3 utilizes the methods described in Chapters 4 - 7 which make use of multiple machine learning classifiers and API frequency analysis to inform the resulting automated analysis referenced in step 4 above. This final analysis step correlates the output of several machine learning classifiers and compares these results to a malware threat classification rubric which can be modified to meet the security posture of a particular organization.

In the following sections we will describe how this final analysis step is performed.

8.3.1 Bespoke Threat Categorization

Below we describe the processes resulting from the use of our framework. This first includes the ingestion of threat-related information delivered as a result of machine learning classification of malware which describes behaviors, factors, or conditions which an organization chooses to base threat-related decisions upon. Next, a threat value is assigned to the malicious software which will align with the risk prioritization of a particular organization.

As this WOPR framework is a system which ultimately would require an organization utilizing it to have a cybersecurity professional actively engaged in its implementation, we will refer to the "end user" of WOPR as such an individual. We assume that this end user has full knowledge of an individual organization's information security risks as well as all mitigating security factors so as to make informed decisions regarding the impact of malware threats with respect to an individual organization.

In Figure 8.3 we present the concept of threat categorization which allows for customization by the end user. Here concern is made regarding the attack patterns associated with a particular malware threat identified as a result of malware family classification. An example of an attack pattern would be the encryption of files located on an infected system. Malware which has been identified as ransomware would likely perform this activity on a victim system.

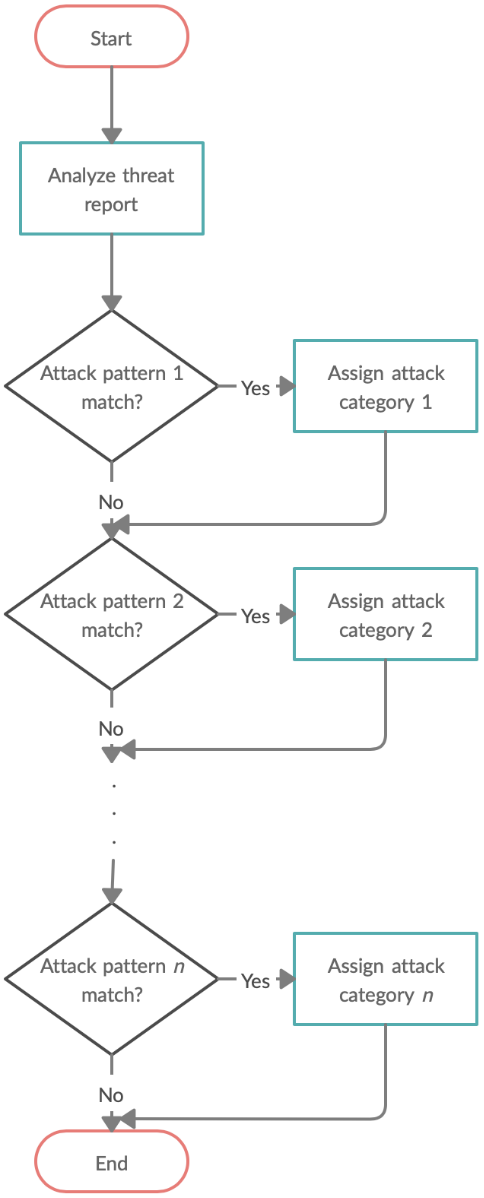


FIGURE 8.3: Flow diagram describing threat category determination.

In another example of malware analysis, an end user may identify “persistence” as an attack pattern they are concerned with, where persistence refers to a set of actions which permit the malware to remain on the system after a reboot or other system change. Many organizations may treat the threat of malware persistence as a critical issue. However, an organization which serves volatile virtual workstations from a cloud server to its users with no expectation of a static environment to store data (such as a public internet cafe) would likely not hold this threat in the same regard.

In previous chapters we have described how malware classification according to a malware family may describe what malicious behaviors to expect, informing this threat categorization process accordingly. Therefore a library of attack patterns can be compiled according to known malware threats as well as previously unknown malware identified through the methodology described in previous chapters. By giving the end user the ability to dynamically define what attack patterns they are concerned with, the output of WOPR will have greater value to the end user than as seen with existing solutions.

8.3.2 Assignment of Threat Value

With defined threat categories assigned to the analyzed malware, our methodology then enters the process which assigns a threat value for an attack category. Figure 8.4 illustrates this process, beginning with an analysis of the assigned threat categories and then comparing a user-defined threat value for each category. For example, an organization which deploys volatile virtual workstations may consider the threat of persistence to be of a “medium” severity because they largely control the risk due to the volatile nature of the workstations, meaning that the changes to the system are not normally saved. However, other organizations that issue physical workstations to their end users may consider persistence to be a “critical” issue due to a lack of security controls on these systems.

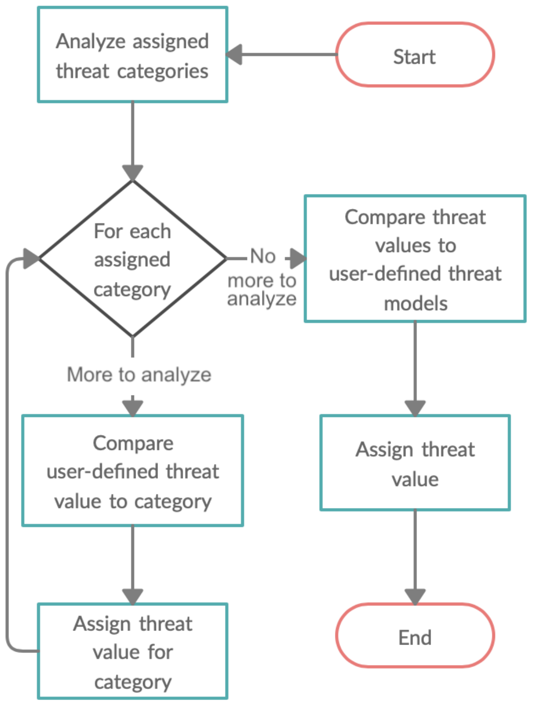

Start
Analyze assigned threat categories
For each assigned category
No more to analyze
More to analyze
Compare threat values to user-defined threat models
Compare user-defined threat value to category
Assign threat value for category
Assign threat value
End

Figure 8.4: Flow diagram describing threat value assignment.

The end user’s ability to assign threat values to attack categories in WOPR’s design borrows from malware threat grading scales such as previously seen in the Infocon rubric. However unlike Infocon, the exact value used to modify a criticality score is configurable by the end user. Therefore, as in the case of the Wannacry ransomware, if the end user for an organization making use of Windows workstations and servers wishes to assign a value of +2 to indicate a critical severity for the detection of file encryption behavior and a +2 to indicate a critical severity for the detection of Windows SMB vulnerability exploitation, then the combined value of 4 for the malware threat may be appropriate for the perceived risk this malware presents to this organization.

Conversely, an end user for an organization with only Linux workstations and servers may feel that a Wannacry threat is of minimal risk. Therefore they may assign a value of +2 for the detection of file encryption behavior to indicate a critical severity, yet perhaps a -1 for the detection of Windows SMB vulnerability exploitation since Windows vulnerabilities are not a concern. This allows for bespoke threat values which the end user decides aligns with their organization's existing mitigating security controls and operating environment.

8.4 Discussion and Recommendations

In this chapter we have identified flaws associated with preconfigured threat assessment systems and methodologies, with particular emphasis on the applicability of the resulting malware criticality score within the context of individual organizations. We have presented a novel bespoke malware risk classification framework which utilizes several beneficial aspects of existing methodologies, such as the use of a severity score rubric and correlation of threat information from several sources, while allowing the risk calculation to take into account mitigating security controls and operating environment of the organization utilizing this framework.

Our bespoke methodology provides an organization with the ability to directly map threats to risks and assign both categorical and impact values in alignment with the unique challenges faced, rather than attempting to bend to a “one size fits all” solution. This methodology combines a simple scoring rubric for malware threats along with an intelligent machine learning approach which is completely transparent to the end user. Implementation of this bespoke methodology will allow organizations to more accurately model threat and risk as a result of the customization allowed under this framework and allows for the identification of potentially unknown risks. Furthermore, this bespoke methodology can be used across a range of industries and its extensibility allows for both user-defined and existing threat classification models.

Chapter 9

Conclusion and Future Research Direction

Malware threats constantly evolve, polymorphically changing code execution sequences and altering behaviors in effort to frustrate detection mechanisms. The resulting arms race between malware authors and cybersecurity professionals has encouraged focus on assessing vulnerabilities in systems and software as a means of disrupting the ability for malware to exploit these vulnerabilities. Frameworks and best practices have been developed to aid in cybersecurity defense, but these efforts routinely ignore malware threats due to their ever-changing nature. This has resulted in fewer tools and techniques to evaluate the severity of a malware threat than the consideration of the impact of vulnerabilities. Tools which do currently exist to assess malware threat are generalized toward the majority of computing systems and environments and while they are better than having nothing at all, their features are not as robust as found in system and software vulnerability assessment tools.

In this dissertation, we assert that bespoke automated malware threat analysis and risk response is more accurate and beneficial to an organization than generalized automated solutions. In support of this assertion, we have developed machine learning methodologies for classifying software as malicious or benign by virtue of operating system API function calls made by software when executed within a computer system. Additionally, our novel methodologies of malware family classification through API frequency analysis as well as analysis of API sequences produces several indicators which an automated system can correlate, thereby increasing confidence in the malware classification performed. An automated malware threat analysis without a bespoke threat evaluation component aims to support the needs of all organizations, and in so doing fails to take into account the differences in security posture each organization has.

However, a bespoke malware threat evaluation suffers from the very attribute which makes it more valuable to an individual organization: subjectivity. A downside of the approach we present lies in the lack of objectivity which would benefit a threat information exchange. If organization A values a ransomware threat as extremely high due to the potential for lost data and organization B devalues the same ransomware threat due to a robust data recovery program, neither of these organizations would be able to agree on the risk impact of such a malware threat. Therefore any malware threat information either organization could share with others would be similarly colored by their faith in their existing security controls, or resulting fear of unmitigated risks due to a lack of mitigating systems or actions.

While the methodologies presented in this dissertation have been focused on malware threat analysis, similar problems exist in the cybersecurity field. One key problem lies in the understanding of how malware threats are related. While this work shows that malware can be classified into families based on API call behavior, further work could explore the relationships between malware families by discovering concrete connections between malware families and variants. In so doing, malware evolution could be modelled and new malware could potentially be predicted from existing code.

Additionally, this dissertation was motivated by a desire to replicate and automate the critical thinking a human cybersecurity analyst might exhibit when performing malware incident response. Malware analysis is but one component of cybersecurity, and the success of the methodologies presented in this work suggest that it could be extended to such activities as software vulnerability assessment, user account behavior anomaly detection, network anomaly detection, and others.

9.1 Peer-Reviewed Publications

Publications that resulted from this dissertation work

[1] Aaron Walker, Muhammad Faisal Amjad, and Shamik Sengupta. Cuckoo's malware threat scoring and classification: Friend or foe? In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0678–0684. IEEE, 2019.

[2] Aaron Walker, Tapadhir Das, Raj Mani Shukla, and Shamik Sengupta. Friend or foe: Discerning benign vs malicious software and malware family. In *2021 IEEE Global Communications Conference: Communication & Information Systems Security (Globecom2021 CISS)*, Madrid, Spain, December 2021.

[3] Aaron Walker and Shamik Sengupta. Insights into malware detection via behavioral frequency analysis using machine learning. In *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2019.

[4] Aaron Walker and Shamik Sengupta. Malware family fingerprinting through behavioral analysis. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–5. IEEE, 2020.

[5] Aaron Walker, Raj Mani Shukla, Tapadhir Das, and Shamik Sengupta. Ohana means family: Malware family classification using extreme learning machines. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC) (CCNC 2022)*, Las Vegas, USA, January 2022.

Appendix A

Machine learning experimentation results

The following section illustrates the extensive results of our experimentation in the form of tables presenting Precision, Recall, and F1-score as a result of MLP networks trained on both malicious and benign software as described in Chapter 6. Table A.1 describes the software used in our experiments. Confusion matrices accompany the tables for each experiment.

A.0.1 Tables

Table A.1: Malware by Antivirus Signature Classification & Benign Software Set

ID	Signature	ID	Benign Executable
M1	Virus:VBS/Ramnit.gen!A	B1	7-Zip 32-bit
M2	Virus:VBS/Ramnit.gen!C	B2	7-Zip 64-bit
M3	PUA:Win32/Puamson.A!ml	B3	Avira Antivirus
M4	TrojanClicker:JS/Faceliker.M	B4	CCleaner
M5	Trojan:JS/Iframeinject	B5	Google Chrome
M6	Trojan:HTML/Redirector.CF	B6	Epson scanner software
M7	Trojan:Win32/Skeeyah.A!bit		
M8	Exploit:HTML/IframeRef.gen	B7	GifCam animated gif software
M9	Virus:VBS/Ramnit.B		
M10	TrojanClicker:JS/Faceliker.D	B8	GIMP image software
M11	TrojanClicker:JS/Faceliker.C		
M12	Trojan:JS/Redirector.QE	B9	OpenVPN
M13	Trojan:JS/BlacoleRef	B10	Ultrasurf proxy
M14	PUA:Win32/Presenoker	B11	Microsoft Visual Studio Code
M15	Trojan:HTML/Brocoiner.D!lib		
M16	TrojanClicker:JS/Faceliker!rfn		
M17	Trojan:Win32/Vibem.O		
M18	Trojan:HTML/Redirector.EP		
M19	Exploit:HTML/IframeRef		
M20	TrojanClicker:JS/Faceliker.A		
M21	Exploit:HTML/IframeRef.DM		
M22	Trojan:HTML/Phish		
M23	PUA:Win32/Kuaiba		

Table A.2: Statistics for Experiment 1.1

	Precision	Recall	F1-score	Support
B1	0.53	0.90	0.67	906
M12	0.59	0.22	0.32	880
M9	1.00	0.89	0.94	914
Accuracy			0.68	2700
Macro Avg.	0.71	0.67	0.64	2700
Weighted Avg.	0.71	0.68	0.65	2700

Table A.3: Confusion Matrix for Experiment 1.1

Predicted Class	Actual Class		
	B1	M12	M9
B1	818	88	0
M12	684	196	0
M9	49	47	818

Table A.4: Statistics for Experiment 1.2

	Precision	Recall	F1-score	Support
M5	0.50	0.41	0.45	914
B2	1.00	0.89	0.94	880
M2	0.51	0.65	0.57	906
Accuracy			0.65	2700
Macro Avg.	0.67	0.65	0.65	2700
Weighted Avg.	0.66	0.65	0.65	2700

Table A.5: Confusion Matrix for Experiment 1.2

Predicted Class	Actual Class		
	M5	B2	M2
M5	374	0	540
B2	64	783	33
M2	317	0	589

Table A.6: Statistics for Experiment 1.3

	Precision	Recall	F1-score	Support
M17	0.98	0.78	0.87	880
B3	0.83	0.86	0.84	914
M9	0.85	1.00	0.92	906
Accuracy			0.88	2700
Macro Avg.	0.89	0.88	0.88	2700
Weighted Avg.	0.89	0.88	0.88	2700

Table A.7: Confusion Matrix for Experiment 1.3

Predicted Class	Actual Class		
	B1	B3	M9
M17	687	155	38
B3	16	782	116
M9	0	0	906

Table A.8: Statistics for Experiment 2.1

	Precision	Recall	F1-score	Support
B4	0.53	0.4	0.46	897
M10	1.00	0.99	1.00	914
M12	0.51	0.63	0.56	889
Accuracy			0.68	2700
Macro Avg.	0.68	0.68	0.67	2700
Weighted Avg.	0.68	0.68	0.68	2700

Table A.9: Confusion Matrix for Experiment 2.1

Predicted Class	Actual Class		
	B4	M10	M12
B4	363	0	534
M10	1	906	7
M12	326	0	563

Table A.10: Statistics for Experiment 2.2

	Precision	Recall	F1-score	Support
B5	0.50	0.98	0.66	897
M11	1.00	0.91	0.95	914
M7	0.57	0.06	0.11	889
Accuracy			0.65	2700
Macro Avg.	0.69	0.65	0.57	2700
Weighted Avg.	0.69	0.65	0.58	2700

Table A.11: Confusion Matrix for Experiment 2.2

Predicted Class	Actual Class		
	B5	M11	M7
B5	881	2	14
M11	54	835	25
M7	837	0	52

Table A.12: Statistics for Experiment 2.3

	Precision	Recall	F1-score	Support
B6	0.48	1.00	0.65	880
M22	0.96	0.43	0.60	896
M18	0.89	0.46	0.61	924
Accuracy			0.63	2700
Macro Avg.	0.78	0.63	0.62	2700
Weighted Avg.	0.78	0.63	0.62	2700

Table A.13: Confusion Matrix for Experiment 2.3

Predicted Class	Actual Class		
	B6	M22	M18
B6	878	0	2
M22	458	389	49
M18	482	18	424

Table A.14: Statistics for Experiment 3.1

	Precision	Recall	F1-score	Support
B9	0.86	0.41	0.55	906
B7	0.62	0.97	0.75	880
B8	0.96	0.93	0.95	914
Accuracy			0.77	2700
Macro Avg.	0.81	0.77	0.75	2700
Weighted Avg.	0.82	0.77	0.75	2700

Table A.15: Confusion Matrix for Experiment 3.1

Predicted Class	Actual Class		
	B9	B7	B8
B9	367	509	30
B7	20	856	4
B8	38	23	853

Table A.16: Statistics for Experiment 3.2

	Precision	Recall	F1-score	Support
B10	0.89	0.71	0.79	840
B4	0.67	0.70	0.68	891
B11	0.56	0.64	0.60	916
Accuracy			0.68	2700
Macro Avg.	0.71	0.68	0.69	2700
Weighted Avg.	0.70	0.68	0.69	2700

Table A.17: Confusion Matrix for Experiment 3.2

Predicted Class	Actual Class		
	B10	B4	B11
B10	593	19	228
B4	36	623	232
B11	39	287	590

Table A.18: Statistics for Experiment 3.3

	Precision	Recall	F1-score	Support
B1	0.50	0.26	0.35	914
B2	0.47	0.98	0.63	906
B3	0.49	0.17	0.25	880
Accuracy			0.47	2700
Macro Avg.	0.48	0.47	0.41	2700
Weighted Avg.	0.48	0.47	0.41	2700

Table A.19: Confusion Matrix for Experiment 3.3

Predicted Class	Actual Class		
	B1	B2	B3
B1	241	517	156
B2	14	890	2
B3	228	503	149

Table A.20: Statistics for Experiment 4.1

	Precision	Recall	F1-score	Support
B8	0.87	0.57	0.69	914
B7	0.70	0.89	0.79	880
M12	0.93	1.00	0.96	906
Accuracy			0.82	2700
Macro Avg.	0.83	0.82	0.81	2700
Weighted Avg.	0.83	0.82	0.81	2700

Table A.21: Confusion Matrix for Experiment 4.1

Predicted Class	Actual Class		
	B8	B7	M12
B1	519	336	59
B7	80	786	14
M12	0	0	906

Table A.22: Statistics for Experiment 4.2

	Precision	Recall	F1-score	Support
B10	0.98	0.21	0.34	916
M9	0.61	0.49	0.54	839
B11	0.50	0.99	0.66	892
Accuracy			0.56	2647
Macro Avg.	0.69	0.56	0.52	2647
Weighted Avg.	0.70	0.56	0.51	2647

Table A.23: Confusion Matrix for Experiment 4.2

Predicted Class	Actual Class		
	B10	M12	B11
B10	190	258	468
M9	3	410	426
B11	1	7	884

Table A.24: Statistics for Experiment 4.3

	Precision	Recall	F1-score	Support
B1	0.50	0.81	0.62	914
M12	0.82	1.00	0.90	906
M9	0.38	0.05	0.10	880
Accuracy			0.63	2700
Macro Avg.	0.57	0.62	0.54	2700
Weighted Avg.	0.57	0.63	0.54	2700

Table A.25: Confusion Matrix for Experiment 4.3

Predicted Class	Actual Class		
	B1	M12	M9
B1	740	96	78
M12	0	906	0
M9	735	97	48

Table A.26: Statistics for Experiment 5.1

	Precision	Recall	F1-score	Support
M9	0.40	0.74	0.52	913
M12	0.35	0.20	0.25	882
M3	0.44	0.26	0.33	905
Accuracy			0.40	2700
Macro Avg.	0.40	0.40	0.37	2700
Weighted Avg.	0.40	0.40	0.37	2700

Table A.27: Confusion Matrix for Experiment 5.1

Predicted Class	Actual Class		
	M9	M12	M3
M9	672	129	112
M12	521	175	186
M3	469	199	237

Table A.28: Statistics for Experiment 5.2

	Precision	Recall	F1-score	Support
M5	0.60	0.19	0.29	913
M23	1.00	0.99	1.00	882
M2	0.51	0.87	0.65	905
Accuracy			0.68	2700
Macro Avg.	0.71	0.69	0.64	2700
Weighted Avg.	0.70	0.68	0.64	2700

Table A.29: Confusion Matrix for Experiment 5.2

Predicted Class	Actual Class		
	M5	M23	M2
M5	174	231	0
M23	437	444	1
M2	5	94	806

Table A.30: Statistics for Experiment 5.3

	Precision	Recall	F1-score	Support
M9	0.61	0.75	0.67	913
M14	0.58	0.50	0.54	882
M17	1.00	0.89	0.94	905
Accuracy			0.72	2700
Macro Avg.	0.73	0.71	0.72	2700
Weighted Avg.	0.73	0.72	0.72	2700

Table A.31: Confusion Matrix for Experiment 5.3

Predicted Class	Actual Class		
	M9	M14	M17
M9	682	231	0
M14	437	444	1
M17	5	94	806

Table A.32: Statistics for Experiment 6.1

	Precision	Recall	F1-score	Support
M6	0.36	0.52	0.42	913
M10	0.37	0.40	0.39	882
M12	0.33	0.15	0.21	905
Accuracy			0.36	2700
Macro Avg.	0.35	0.36	0.34	2700
Weighted Avg.	0.35	0.36	0.34	2700

Table A.33: Confusion Matrix for Experiment 6.1

Predicted Class	Actual Class		
	M6	M10	M12
M6	473	304	136
M10	386	353	143
M12	472	293	140

Table A.34: Statistics for Experiment 6.2

	Precision	Recall	F1-score	Support
M7	0.38	0.03	0.06	913
M11	0.41	0.50	0.45	882
M15	0.37	0.63	0.47	905
Accuracy			0.39	2700
Macro Avg.	0.39	0.39	0.33	2700
Weighted Avg.	0.39	0.39	0.32	2700

Table A.35: Confusion Matrix for Experiment 6.2

Predicted Class	Actual Class		
	M7	M11	M15
M7	30	311	572
M11	37	441	404
M15	13	319	573

Table A.36: Statistics for Experiment 6.3

	Precision	Recall	F1-score	Support
M22	0.36	0.43	0.40	913
M13	0.33	0.61	0.43	882
M18	0.00	0.00	0.00	905
Accuracy			0.34	2700
Macro Avg.	0.23	0.35	0.27	2700
Weighted Avg.	0.23	0.34	0.27	2700

Table A.37: Confusion Matrix for Experiment 6.3

Predicted Class	Actual Class		
	M22	M13	M18
M22	394	519	0
M13	348	534	0
M18	338	567	0

Table A.38: Statistics for Experiment 7.1

	Precision	Recall	F1-score	Support
M1	1.00	0.96	0.98	913
M9	0.55	0.74	0.63	882
M2	0.61	0.42	0.50	905
Accuracy			0.71	2700
Macro Avg.	0.72	0.71	0.70	2700
Weighted Avg.	0.72	0.71	0.70	2700

Table A.39: Confusion Matrix for Experiment 7.1

Predicted Class	Actual Class		
	M1	M9	M2
M1	876	18	19
M9	0	655	227
M2	0	521	384

Table A.40: Statistics for Experiment 7.2

	Precision	Recall	F1-score	Support
M4	0.34	0.29	0.31	913
M16	0.32	0.42	0.36	882
M20	0.32	0.27	0.29	905
Accuracy			0.32	2700
Macro Avg.	0.33	0.32	0.32	2700
Weighted Avg.	0.33	0.32	0.32	2700

Table A.41: Confusion Matrix for Experiment 7.2

Predicted Class	Actual Class		
	M4	M16	M20
M4	262	397	254
M16	247	371	264
M20	256	408	241

Table A.42: Statistics for Experiment 7.3

	Precision	Recall	F1-score	Support
M21	0.34	0.98	0.51	913
M8	0.35	0.03	0.06	882
M19	0.00	0.00	0.00	905
Accuracy			0.34	2700
Macro Avg.	0.23	0.34	0.19	2700
Weighted Avg.	0.23	0.34	0.19	2700

Table A.43: Confusion Matrix for Experiment 7.3

Predicted Class	Actual Class		
	M21	M8	M19
M21	894	19	0
M8	853	29	0
M19	871	34	0

Bibliography

[1] Michele Maasberg, Myung Ko, and Nicole L Beebe. Exploring a systematic approach to malware threat assessment. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5517–5526. IEEE, 2016.

[2] Forum of Incident Response and Inc. Security Teams. Cvss v3.1 user guide, Jun 2019. URL `https://www.first.org/cvss/user-guide`.

[3] Oskars Podzins and Andrejs Romanovs. Why siem is irreplaceable in a secure it environment? In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2019.

[4] MGMUA Entertainment Company, May 1983.

[5] Charles Feng, Shuning Wu, and Ningwei Liu. A user-centric machine learning framework for cyber security operations center. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 173–175. IEEE, 2017.

[6] Ramandika Pranamulia, Yudistira Asnar, and Riza Satria Perdana. Profile hidden markov model for malware classification—usage of system call sequence for malware classification. In *2017 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–5. IEEE, 2017.

[7] Domhnall Carlin, Alexandra Cowan, Philip O'kane, and Sakir Sezer. The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes. *IEEE Access*, 5:17742–17752, 2017.

[8] Michael Howard, Avi Pfeffer, Mukesh Dalai, and Michael Reposa. Predicting signatures of future malware variants. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 126–132. IEEE, 2017.

[9] Qiang Liu, Pan Li, Wentao Zhao, Wei Cai, Shui Yu, and Victor CM Leung. A survey on security threats and defensive techniques of machine learning: A data driven view. *IEEE access*, 6:12103–12117, 2018.

[10] Bowen Sun, Qi Li, Yanhui Guo, Qiaokun Wen, Xiaoxi Lin, and Wenhan Liu. Malware family classification method based on static feature extraction. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 507–513. IEEE, 2017.

[11] Pengbin Feng, Jianfeng Ma, Cong Sun, Xinpeng Xu, and Yuwan Ma. A novel dynamic android malware detection system with ensemble learning. *IEEE Access*, 6:30996–31011, 2018.

[12] Pedro Casas, Gonzalo Marín, Germán Capdehourat, and Maciej Korczynski. Mlsec-benchmarking shallow and deep machine learning models for network security. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 230–235. IEEE, 2019.

[13] Giovanni Apruzzese, Michele Colajanni, Luca Ferretti, Alessandro Guido, and Mirco Marchetti. On the effectiveness of machine and deep learning for cyber security. In *2018 10th International Conference on Cyber Conflict (CyCon)*, pages 371–390. IEEE, 2018.

[14] Kaixing Huang, Chunjie Zhou, Yu-Chu Tian, Weixun Tu, and Yuan Peng. Application of bayesian network to data-driven cyber-security risk assessment in scada networks. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6. IEEE, 2017.

[15] Prashanth Krishnamurthy, Ramesh Karri, and Farshad Khorrami. Anomaly detection in real-time multi-threaded processes using hardware performance counters. *IEEE Transactions on Information Forensics and Security*, 15:666–680, 2019.

[16] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[17] Xinjian Ma, Qi Biao, Wu Yang, and Jianguo Jiang. Using multi-features to reduce false positive in malware classification. In *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, pages 361–365. IEEE, 2016.

[18] Aziz Alotaibi. Identifying malicious software using deep residual long-short term memory. *IEEE Access*, 7:163128–163137, 2019.

[19] Tianshi Mu, Huajun Chen, Jinran Du, and Aidong Xu. An android malware detection method using deep learning based on api calls. In *2019 IEEE 3rd Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 2001–2004. IEEE, 2019.

[20] Di Xue, Jingmei Li, Tu Lv, Weifei Wu, and Jiaxiang Wang. Malware classification using probability scoring and machine learning. *IEEE Access*, 7:91641–91656, 2019.

[21] Blake E Strom, Joseph A Battaglia, Michael S Kemmerer, William Kupersanin, Douglas P Miller, Craig Wampler, Sean M Whitley, and Ross D Wolf. Finding cyber threats with att&ck-based analytics. *The MITRE Corporation, Tech. Rep.*, 2017.

[22] Automated malware analysis, Feb 2014. URL `http://www.cuckoosandbox.org/`.

[23] Michael Smith, Joey Ingram, Christopher Lamb, Timothy Draelos, Justin Doak, James Aimone, and Conrad James. Dynamic analysis of executables to detect and characterize malware. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 16–22. IEEE, 2018.

[24] Hajredin Daku, Pavol Zavarsky, and Yasir Malik. Behavioral-based classification and identification of ransomware variants using machine learning. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1560–1564. IEEE, 2018.

[25] Abdurrahman Pektaş and Tankut Acarman. Malware classification based on api calls and behaviour analysis. *IET Information Security*, 12(2):107–117, 2018.

[26] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Zero-day malware detection. In *2016 Sixth international symposium on embedded computing and system design (ISED)*, pages 171–175. IEEE, 2016.

[27] SL Shiva Darshan, MA Ajay Kumara, and CD Jaidhar. Windows malware detection based on cuckoo sandbox generated report using machine learning algorithm. In *2016 11th International Conference on Industrial and Information Systems (ICIIS)*, pages 534–539. IEEE, 2016.

[28] Qian Chen and Robert A Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 454–460. IEEE, 2017.

[29] Aruna Jain and Akash Kumar Singh. Integrated malware analysis using machine learning. In *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, pages 1–8. IEEE, 2017.

[30] Jhu-Sin Luo and Dan Chia-Tien Lo. Binary malware image classification using machine learning with local binary pattern. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4664–4667. IEEE, 2017.

[31] Steven Strandlund Hansen, Thor Mark Tampus Larsen, Matija Stevanovic, and Jens Myrup Pedersen. An approach for detection and family classification of malware based on behavioral analysis. In *2016 International conference on computing, networking and communications (ICNC)*, pages 1–5. IEEE, 2016.

[32] Robert Luh, Sebastian Schrittwieser, and Stefan Marschalek. Llr-based sentiment analysis for kernel event sequences. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 764–771. IEEE, 2017.

[33] Ehab Mufid Shafiq Alkhateeb. Dynamic malware detection using api similarity. In *2017 IEEE International Conference on Computer and Information Technology (CIT)*, pages 297–301. IEEE, 2017.

[34] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[35] Joakim Kargaard, Tom Drange, Ah-Lian Kor, Hissam Twafik, and Emlyn Butterfield. Defending it systems against intelligent malware. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 411–417. IEEE, 2018.

[36] Jing Liu, Yuan Wang, and Yongjun Wang. Inferring phylogenetic networks of malware families from api sequences. In *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 14–17. IEEE, 2016.

[37] Trung Kien Tran and Hiroshi Sato. Nlp-based approaches for malware classification from api sequences. In *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, pages 101–105. IEEE, 2017.

[38] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[39] Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.

[40] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[41] Aran Lakhotia and Paul Black. Mining malware secrets. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–18. IEEE, 2017.

[42] Chunlei Zhao, Wenbai Zheng, Liangyi Gong, Mengzhe Zhang, and Chundong Wang. Quick and accurate android malware detection based on sensitive apis. In *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 143–148. IEEE, 2018.

[43] Michael Brunner, Christian Sillaber, and Ruth Breu. Towards automation in information security management systems. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 160–167. IEEE, 2017.

[44] Michael Mylrea, Sri Nikhil Gupta Gourisetti, Curtis Larimer, and Christine Noonan. Insider threat cybersecurity framework webtool & methodology: Defending against complex cyber-physical threats. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 207–216. IEEE, 2018.

[45] Adam Sedgewick. Framework for improving critical infrastructure cybersecurity, version 1.0. Technical report, 2014.

[46] Sri Nikhil Gupta Gourisetti, Michael Mylrea, Travis Ashley, Roger Kwon, Jerry Castleberry, Quinn Wright-Mockler, Penny McKenzie, and Geoffrey Brege. Demonstration of the cybersecurity framework through real-world cyber attack. In *2019 Resilience Week (RWS)*, volume 1, pages 19–25. IEEE, 2019.

[47] M Ugur Aksu, M Hadi Dilek, E İslam Tatlı, Kemal Bicakci, H Ibrahim Dirik, M Umut Demirezen, and Tayfun Aykır. A quantitative cvss-based cyber security risk assessment methodology for it systems. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8. IEEE, 2017.

[48] Taehoon Eom, Jin B Hong, Seongmo An, Jong Sou Park, and Dong Seong Kim. A systematic approach to threat modeling and security analysis for software defined networking. *Ieee Access*, 7:137432–137445, 2019.

[49] Ngoc T Le and Doan B Hoang. Security threat probability computation using markov chain and common vulnerability scoring system. In *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6. IEEE, 2018.

[50] Aakarsh Rao, Nadir Carreón, Roman Lysecky, and Jerzy Rozenblit. Probabilistic threat detection for risk management in cyber-physical medical systems. *IEEE Software*, 35(1):38–43, 2017.

[51] Daniel M Best, Jaspreet Bhatia, Elena S Peterson, and Travis D Breaux. Improved cyber threat indicator sharing by scoring privacy risk. In *2017 IEEE*

International Symposium on Technologies for Homeland Security (HST), pages 1–5. IEEE, 2017.

[52] Roland Meier, Cornelia Scherrer, David Gugelmann, Vincent Lenders, and Laurent Vanbever. Feedrank: A tamper-resistant method for the ranking of cyber threat intelligence feeds. In *2018 10th International Conference on Cyber Conflict (CyCon)*, pages 321–344. IEEE, 2018.

[53] Victor R Kebande, Ivans Kigwana, HS Venter, Nickson M Karie, and Ruth D Wario. Cvss metric-based analysis, classification and assessment of computer network threats and vulnerabilities. In *2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, pages 1–10. IEEE, 2018.

[54] Holman Bolívar, Héctor Dario Jaimes Parada, Olga Roa, and John Velandia. Multi-criteria decision making model for vulnerabilities assessment in cloud computing regarding common vulnerability scoring system. In *2019 Congreso Internacional de Innovación y Tendencias en Ingenieria (CONIITI)*, pages 1–6. IEEE, 2019.

[55] Pontus Johnson, Robert Lagerström, Mathias Ekstedt, and Ulrik Franke. Can the common vulnerability scoring system be trusted? a bayesian analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):1002–1015, 2016.

[56] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[57] Nur Syuhada Selamat, Fakariah Hani Mohd Ali, and Noor Ashitah Abu Othman. Polymorphic malware detection. In *2016 6th International Conference on IT Convergence and Security (ICITCS)*, pages 1–5. IEEE, 2016.

[58] Preetam Mukherjee and Chandan Mazumdar. “security concern” as a metric for enterprise business processes. *IEEE Systems Journal*, 13(4):4015–4026, 2019.

[59] Denis Chernov and Alexey Sychugov. Development of a mathematical model of threat to information security of automated process control systems. In *2019 International Russian Automation Conference (RusAutoCon)*, pages 1–5. IEEE, 2019.

[60] Regner Sabillon, Jordi Serra-Ruiz, Victor Cavaller, and Jeimy Cano. A comprehensive cybersecurity audit model to improve cybersecurity assurance: The cybersecurity audit model (csam). In *2017 International Conference on Information Systems and Computer Science (INCISCOS)*, pages 253–259. IEEE, 2017.

[61] Richard A Caralli, James F Stevens, Lisa R Young, and William R Wilson. Introducing octave allegro: Improving the information security risk assessment process. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2007.

[62] Anisa Dewi Prajanti and Kalamullah Ramli. A proposed framework for ranking critical information assets in information security risk assessment using the

octave allegro method with decision support system methods. In *2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1–4. IEEE, 2019.

[63] Khangwelo Muronga, Marlein Herselman, Adele Botha, and Adéle Da Veiga. An analysis of assessment approaches and maturity scales used for evaluation of information security and cybersecurity user awareness and training programs: A scoping review. In *2019 Conference on Next Generation Computing Applications (NextComp)*, pages 1–6. IEEE, 2019.

[64] Daniel Qi Chen and Huigang Liang. Wishful thinking and it threat avoidance: An extension to the technology threat avoidance theory. *IEEE Transactions on Engineering Management*, 66(4):552–567, 2019.

[65] Rui Azevedo, Ibéria Medeiros, and Alysson Bessani. Pure: Generating quality threat intelligence by clustering and correlating osint. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 483–490. IEEE, 2019.

[66] Ghaith Husari, Xi Niu, Bill Chu, and Ehab Al-Shaer. Using entropy and mutual information to extract threat actions from cyber threat intelligence. In *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6. IEEE, 2018.

[67] Joshua Cannell and Joshua Cannell. Automating malware analysis with cuckoo sandbox, Apr 2016. URL `https://blog.malwarebytes.com/threat-analysis/2014/04/automating-malware-analysis-with-cuckoo-sandbox/`.

[68] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape. *Proceedings of the Botconf*, 2017.

[69] Cuckoosandbox. Really low scores for hardcore malware · issue #2019 · cuckoosandbox/cuckoo, . URL `https://github.com/cuckoosandbox/cuckoo/issues/2019#issuecomment-352305821`.

[70] Cuckoosandbox. Magic numbers in signatures score · issue #732 · cuckoosandbox/cuckoo, . URL `https://github.com/cuckoosandbox/cuckoo/issues/732#issuecomment-174168657`.

[71] John Kennedy, Michael Satran, and Mark LeBlanc. Api index-windows applications. *Windows Applications-Microsoft Docs*, 2018.

[72] Windows-Sdk-Content. Getasynckeystate function (winuser.h) - win32 apps, Dec 2018. URL `https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getasynckeystate`.

[73] Windows-Sdk-Content. Getcursorpos function (winuser.h) - win32 apps, Dec 2018. URL `https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getcursorpos`.

[74] Will Koehrsen. How to visualize a decision tree from a random forest in python using scikit-learn, Aug 2018. URL `https://tinyurl.com/2txtkymv`.

[75] Jason Brownlee. Your first machine learning project in python step-by-step, Aug 2019. URL `https://machinelearningmastery.com/machine-learning-in-python-step-by-step/`.

[76] Aaron Walker, Muhammad Faisal Amjad, and Shamik Sengupta. Cuckoo's malware threat scoring and classification: Friend or foe? In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0678–0684. IEEE, 2019.

[77] Aaron Walker and Shamik Sengupta. Insights into malware detection via behavioral frequency analysis using machine learning. In *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2019.

[78] Microsoft. Virus:vbs/ramnit.gen!a, Aug 2011. URL `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus:VBS/Ramnit.gen`.

[79] Microsoft. Virus:vbs/ramnit.gen!c, Nov 2011. URL https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus:VBS/Ramnit.gen.

[80] Microsoft. Trojandownloader:win32/vigorf.a, Jun 2016. URL https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Vigorf.A.

[81] Maureen Reyes. Worm.js.bondat.ac - threat encyclopedia, Dec 2018. URL https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/worm.js.bondat.ac.

[82] ORACLE. Oracle vm virtualbox, 2019. URL https://www.virtualbox.org/.

[83] J-Michael Roberts. Virusshare.com. URL https://virusshare.com.

[84] Igor Pavlov. 7-zip, 2019. URL https://www.7-zip.org/.

[85] Avira. Avira antivirus, 2019. URL https://www.avira.com/.

[86] Piriform. Ccleaner official website, Oct 2019. URL https://www.ccleaner.com/.

[87] Google. Google chrome, 2017. URL https://www.google.com/chrome/.

[88] Epson. Epson drivers, 2021. URL https://ftp.epson.com/drivers/ESU_451.exe.

[89] Isa Ali. Bahraniapps blog, Nov 2020. URL `http://blog.bahraniapps.com/`.

[90] GIMP Team. Gimp, 2019. URL `https://www.gimp.org/`.

[91] OpenVPN. Openvpn, 2018. URL `https://openvpn.net/`.

[92] UltraReach. About ultrasurf and ultrareach - internet freedom, privacy, and security, 2021. URL `https://ultrasurf.us/`.

[93] Microsoft. Visual studio code, Apr 2016. URL `https://code.visualstudio.com/`.

[94] Malwarebytes, Feb 2021. URL `https://resources.malwarebytes.com/files/2021/02/MWB_StateOfMalwareReport2021.pdf`.

[95] Terrence August, Duy Dao, and Marius Florin Niculescu. Economics of ransomware attacks. *Earlier Version Presented at WISE*, 2017.

[96] Aaron Walker and Shamik Sengupta. Insights into malware detection via behavioral frequency analysis using machine learning. In *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, pages 1–6, 2019. doi: 10.1109/MILCOM47813.2019.9021034.

[97] Aaron Walker and Shamik Sengupta. Malware family fingerprinting through behavioral analysis. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–5, 2020. doi: 10.1109/ISI49825.2020.9280529.

[98] Aaron Walker, Muhammad Faisal Amjad, and Shamik Sengupta. Cuckoo's malware threat scoring and classification: Friend or foe? In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0678–0684, 2019. doi: 10.1109/CCWC.2019.8666454.

[99] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[100] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang. Extreme learning machine for regression and multiclass classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):513–529, 2011.

[101] Jin-Man Park and Jong-Hwan Kim. Online recurrent extreme learning machine and its application to time-series prediction. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1983–1990. IEEE, 2017.

[102] Nan-Ying Liang, Guang-Bin Huang, Paramasivan Saratchandran, and Narasimhan Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Transactions on neural networks*, 17 (6):1411–1423, 2006.

[103] C Radhakrishna Rao. Generalized inverse of a matrix and its applications. In *Vol. 1 Theory of Statistics*, pages 601–620. University of California Press, 1972.

[104] ISACA. Cmmi institute - cmmi v2.0, 2019. URL `https://cmmiinstitute.com/cmmi`.

[105] Ron Ross, Victoria Pillitteri, Kelley Dempsey, Mark Riddle, and Gary Guissanie. Protecting controlled unclassified information in nonfederal systems and organizations. Technical report, National Institute of Standards and Technology, 2019.

[106] ISO 27000 Directory. Iso 27000 - iso 27001 and iso 27002 standards, 2021. URL `https://www.27000.org/`.

[107] SANS Internet Storm Center. Infocon. URL `https://isc.sans.edu/infocon.html`.

[108] Inc. WatchGuard Technologies. About tdr threat scores. URL `https://www.watchguard.com\/help\/docs\/help-center\/en-US\/Content\/en-US\/Fireware\/services\/tdr\/tdr_threat_scores\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{h\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\def{}\endgroup\relax\let\ignorespaces\relax\accent95h\egroup\spacefactor\accent@spacefactortml`.

[109] National Institute of Standards and Technology. Nvd - cvss v3 calculator. URL `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

[110] Johannes Ullrich. Wannacry/wannacrypt ransomware summary, May 2017. URL `https://isc.sans.edu/forums/diary/WannaCryWannaCrypt+Ransomware+Summary/22420/`.

[111] Ellen Nakashima and Craig Timberg. Nsa officials worried about the day its potent hacking tool would get loose. then it did. *The Washington Post*, May 2017. URL `https://tinyurl.com/2p889yk8`.