IMPROVING THE EFFECTIVENESS AND EFFICIENCY OF DYNAMIC MALWARE ANALYSIS USING MACHINE LEARNING

by

Leonardo De La Rosa

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Financial Services Analytics

Summer 2018

IMPROVING THE EFFECTIVENESS AND EFFICIENCY OF DYNAMIC MALWARE ANALYSIS USING MACHINE LEARNING

by

Leonardo De La Rosa

Approved: _____
Bintong Chen, Ph.D.
Chair of the Department of Financial Services Analytics

Approved: _____
Bruce Weber, Ph.D.
Dean of the College of Business and Economics

Approved: _____
Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Adam Fleischhacker, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Starnes Walker, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Michael Silas, Ph.D.
Member of dissertation committee

DEDICATION

I would like to dedicate this dissertation to my Mom, my Angel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The malware threat landscape is constantly evolving, with upwards of one million new variants being released every day. Traditional approaches for detecting and classifying malware usually contain brittle handcrafted heuristics that quickly become outdated and can be exploited by nefarious actors. As a result, it is necessary to change the way software security is managed by using advanced analytics (i.e., machine learning) and significantly more automation to develop adaptable malware analysis engines that correctly identify, categorize, and characterize malware.

In this dissertation, we introduce a next-generation sandbox that leverages machine learning to create an adaptive malware analysis platform. This intelligent environment considerably extends the capabilities of Cuckoo, an open-source malware analysis sandbox, and significantly optimizes the resources dedicated to the dynamic analysis of malware.

Dynamic analysis allows security analysts to collect information about the behavior of malicious samples in an isolated environment. However, running malware in a sandbox is time-consuming and computationally expensive. This technique extracts information from malware without executing it and is orders of magnitude faster than dynamic analysis. Nevertheless, for some malware it may still be necessary to use dynamic-based features to produce better classifications and characterizations.

With our system, we were successful in identifying the simplest characterizations required to accurately classify malware. This is an important feature because it allows us to determine the subset of samples that is truly different, and requires very expensive dynamic characterization. When dynamic analysis is imperative, our system also estimates the minimum amount of time required to accurately detect and classify malware. As a result, our intelligent analysis platform can reallocate the time saved to

analyzing files that require longer execution times and produce actionable intelligence for our system. Finally, by leveraging the speed of static analysis, our system induces highly accurate machine learning models for malware capability detection, removing the need to perform dynamic analysis to identify high-level functionalities of malicious code.

Chapter 1

INTRODUCTION

1.1 My Thesis

The central thesis of this dissertation is: we can build an adaptive malware analysis system that leverages machine learning and 1) predicts the length of time needed to execute malware in an isolated environment, 2) learns when dynamic behavior improves the accuracy of malware classification beyond performing static analysis, and 3) automatically identifies the important capabilities of malware.

1.2 Problem Statement

Malicious software, or malware, is a type of computer program designed to cause harm to a user's computer, mobile phone, website or data [Lemonnier, 2015]. Depending on its propagation method and the goal for which it is created, malware can be classified as viruses, worms, trojans, spyware, and ransomware, among other categories [Christodorescu and Jha, 2003]. Malware has evolved from small malicious applications, whose main intent was to expose security vulnerabilities and flaunt the technical ability of its authors [Egele et al., 2012], to complex and sophisticated programs whose goal is to profit from forced advertising (adware), to steal private information (spyware) or to extort money (ransomware) [Symantec Corporation, 2017a]. Furthermore, bad authors have embraced automation, and it has been estimated that over one million new malware are released to the public every day [Harrison and Pagliery, 2015a].

Numerous techniques have been developed over the last three decades to fight the increasing threat of malware. One such technique is a signature-based approach that assigns a signature or unique identifier (i.e., a hash) to each malware sample and attempts to identify the presence of malware on a system by comparing these

hashes against a repository of signatures of known malware [Mujumdar et al., 2013]. Nevertheless, this tool fails at detecting new malware variants for which a signature has not yet been generated [Pandey and Mehtre, 2014].

Heuristic-based tools use rules to examine suspicious files and classify them as malware [Cobb, 2016]. This approach is limited, however, due to the fact that it relies on the appearance of repeated code that is indicative of malicious intent [Cade, 2017].

Static analysis has the ability to extract code from a malicious file without executing the malware binary [Egele et al., 2012]. In addition, static techniques observe the entire structure of malware and characterize all possible execution paths of a malicious sample [Nath and Mehtre, 2014]. Yet, static analysis is hindered by obfuscation techniques such as packing (i.e., programs that transform a malware binary into another form, without affecting its execution semantics [Guo et al., 2008]) and encryption which are designed to hamper the disassembly process of malicious code.

On the other hand, dynamic analysis has emerged as a state-of-the-art approach for malware detection and classification that overcomes the shortcomings of the previously mentioned techniques [Hu et al., 2013]. In this type of analysis, a malicious file is executed in an isolated chamber (or sandbox) and its behavior is monitored and reported for additional examination [Cuckoo, 2017]. The main objective of this type of analysis is to collect information about the behavior of the monitored samples, which can later be used to identify new malicious files [Vadrevu and Perdisci, 2016]. Sandboxes can inspect and record the behavior of any file, including Windows executables, Portable Document Format (PDF) files, Java applets, Microsoft Office documents, mobile phone apps, and malicious websites [Cuckoo, 2017]. In addition, by executing malicious software in a sandbox, it is possible to construct behavioral profiles that describe the high-level functionalities or ***capabilities*** of malware, such as the ability to detect the presence of an analysis environment or capture key strokes [Saxe et al., 2014].

Sandboxes have become an excellent cybersecurity tool in the security analyst's arsenal and have gained in popularity over the last decade [Oktavianto and

Muhardianto, 2013], [Blasing et al., 2010]. The growing number of new malware being released daily requires high performance analysis systems that can examine as many malicious files as possible within a short timeframe [Vadrevu and Perdisci, 2016].

Nevertheless, traditional sandbox environments contain many handcrafted heuristics that are inherently brittle, as a small change or mutation to the code of the malware can render them useless [Cade, 2017]. Furthermore, bad actors have developed anti-sandbox malware that will not execute any malicious activity if it detects that it is running in a sandbox environment [Kirat et al., 2011]. By exploiting the side-effects imposed by virtualized systems, malware can inspect the names of the available devices, registry keys, background processes, and IP addresses to detect the presence of the sandbox and change its behavior to circumvent the analysis [Issa, 2012].

Additionally, running malware in a sandboxed environment can be computationally expensive compared to static analysis, and often there is no additional benefit to executing a suspicious file in a sandbox beyond performing fast static analysis of malicious code [Kilgallon et al., 2017].

1.2.1 Need to Quickly Detect Malware in Real Time

Dynamic analysis can be a bottleneck for analyzing malware and deducing its important capabilities [Rhode et al., 2018]. In traditional sandbox environments, malware is examined for a fixed amount of time (e.g., one minute, [Kirat et al., 2011], two minutes [Willems et al., 2007], three minutes [Hansen et al., 2016], four minutes [Christodorescu et al., 2008], or even five minutes [Damodaran et al., 2017], [Tobiyama et al., 2016],[Christodorescu et al., 2008]. Dynamic analysis is performed for long periods of time regardless of the information extracted from the file in question [Bayer et al., 2010], or until the execution of the malware ends [Bayer et al., 2010].

However, a significant amount of malware samples that are submitted for analysis are repackaged versions of previously seen and analyzed malware [Vadrevu and Perdisci, 2016]. In addition, the overall run time might be longer, because numerous systems allow a post-processing phase following the analysis of the binary sample

[Bayer et al., 2010]. This results in a large amount of time being wasted in studying files that do not improve the intelligence of the analysis system.

Moreover, running malware in a sandbox for long periods of time lead to predictive models that cannot scale to millions of malware [Bierma et al., 2014]. As a result, dynamic analysis cannot be efficiently used to construct behavioral profiles and deduce the important capabilities of extremely large malware datasets [Storlie et al., 2014].

Given the increasing stream of new malware that is generated every day, it is necessary to develop techniques that quickly detect malware in real time [Harrison and Pagliery, 2015a]. By constructing models that predict the shortest amount of dynamic analysis time required by each malware, it is possible to reduce the resources spent executing files whose harmful nature can be revealed in seconds [Vadrevu and Perdisci, 2016]. The time saved can then be reallocated to running binaries that demand longer periods of time and produce actionable intelligence for the analysis system [Bayer et al., 2010].

In this dissertation, we introduce an intelligent malware analysis system that leverages machine learning to detect and classify new malware in a short amount of time and deduce its important capabilities. It also optimizes the resources dedicated to the dynamic analysis of malware. Additionally, our system identifies the simplest characterizations required to accurately classify malware. This allows us to determine the subset of samples that is truly different and requires very expensive dynamic characterization. When dynamic analysis is necessary, our system predicts the shortest amount of time required to accurately detect and classify malware. This is a crucial quality because the traditional process of dynamic analysis can be a bottleneck for studying malware [Rhode et al., 2018]. It is therefore essential that our malware analysis system spends the least amount of time analyzing malware whose malicious nature can be detected in seconds [Rhode et al., 2018]. Finally, we leverage the speed of static analysis features [LMSecurity, 2017] to induce highly accurate machine learning models for malware capability detection.

1.3 Organization

We organize the remainder of this dissertation as follows. Chapter 2 provides background information. In the first part, we introduce a historical overview of malicious software and malware analysis techniques. Specifically, we cover signature-based approaches, heuristic-based tools, static and dynamic analysis, as well as the software and techniques that we use to extract dynamic and static features from our malware dataset. These features are used as input to our machine learning algorithms that predict important capabilities of malware, the time needed to execute malicious samples in the sandbox environment, and the simplest and least expensive characterizations that lead to accurate malware family classification models. Additionally, we include a literature review of machine learning for cyber security in order to understand the advances that have been made in the field of malware analysis detection and classification. Finally, we conclude the chapter with a discussion of the limitations of existing techniques that leverage machine learning for malware detection and classification, as well as the approaches with which we move the state-of-the-art forward in dynamic malware analysis and sandbox technology.

Chapter 3 introduces the three machine learning-based approaches proposed in this dissertation. Chapter 4 introduces TURACO, a model used to estimate the approximate amount of time needed to perform dynamic analysis on a file in order to detect its malicious intent. Chapter 5 presents SEEMA, a model that determines when running malware in a sandbox provides additional benefits beyond performing quick and effective static analysis. Chapter 6 describes MAGIC, a model that predicts important capabilities of malware. Chapter 7 concludes this dissertation and Chapter 8 suggests directions and implications for future work.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

2.1 Introduction

In this section, we introduce the background necessary to understand the research described in this dissertation. First, we present a historical overview of malicious software and the techniques that have been proposed to detect, classify and predict important capabilities of malware. Specifically, we describe signature-based approaches, heuristic-based tools, and static and dynamic analysis. In addition, we introduce Cuckoo Sandbox, the malware analysis system used to extract behavioral information from our malware dataset and Radare2, an open-source reverse engineering compiler that can disassemble the executable code of malware binaries. This is followed by a literature review of machine learning for cyber security that discusses the advances that have been made in the field of malware analysis. The chapter is then concluded with a discussion of the limitations of current approaches for malware detection and classification, as well as the techniques with which we move the state-of-the-art forward in dynamic malware analysis and sandbox technology.

2.2 Malware Overview

A computer program designed to cause harm to a user's computer, website or information is commonly referred to as malicious software, or malware [Moser et al., 2007]. Malware can infect computers and devices in numerous ways and comes in a variety of forms. Depending on its propagation method and the intent for which it is created [McGraw and Morrisett, 2000], malicious code is usually classified in the following categories:

1. **Viruses:** correspond to programs that self-replicate throughout a computer or network. Viruses attach themselves to existing programs and documents and can corrupt or delete data, reformat hard disks, and use up computer memory [Paloalto Networks, 2016].

2. **Worms:** are self-replicating viruses that spread across computers and networks. Unlike many viruses, worms do not corrupt files or attach themselves to existing programs. Due to its replication nature, they can consume significant system resources, taking up a lot of space in the hard drive and exhausting network bandwidth [Swain, 2009].

3. **Trojans:** unlike viruses and worms, trojans do not self-replicate [Paloalto Networks, 2016]. They disguise themselves as genuine files or software to create a backdoor in a computer or network, allowing personal and confidential information to be stolen [Swain, 2009].

4. **Ransomware:** is a type of malware that can lock down a machine, encrypt its data, and threaten to erase its contents unless a ransom is paid [Lemonnier, 2015].

5. **Spyware:** refers to a type of malware that gathers information about the usage and browsing habits of the infected machine and sends it back to the attacker. This occurs with or without the permission of the user and includes malicious programs such as botnets, adware, keyloggers, and backdoors, among others [Paloalto Networks, 2016].

Historically, viruses were the first instances of malicious code to appear [Landesman, 2017]. The first computer virus was named Elk Cloner and was found on a Mac in 1982 [GData Internet Security, 2017]. The propagation of most viruses was largely due to infected floppy disks used across college campuses. In addition, the intent of the authors of such early malware was to expose security vulnerabilities and flaunt technical ability [Egele et al., 2012]. By the mid 1990s, businesses were facing significant threats from the appearance of macro viruses (i.e., a type of virus that infects Word, Excel and PowerPoint files [Swain, 2009]) [Landesman, 2017]. By the end of twentieth century, home users were also having severe problems with viruses, as their propagation by email became commonplace [Radware, 2017]. Furthermore, the start of the new millennium saw the rise of internet and email worms, as broadband internet adoption increased [Khanse, 2014].

As time passed, the motivations that drove cybercriminals to create malware changed radically, and now there is a thriving underground economy based on malware [Moser et al., 2007]. It is no longer the mischief factor or vandalism what motivates bad authors to create malware, but the financial gain that can be made from the target user or system [Symantec Corporation, 2017b]. Today, a significant number of malware is developed to profit from forced advertising (adware), to compromise private information (spyware), or to extort money (ransomware) [Symantec Corporation, 2017a].

Furthermore, it has been estimated that upwards of one million malware variants are released every day [Harrison and Pagliery, 2015a]. The alarming rate at which these malware are created is due to the widespread adoption of automated tools, which enable the construction of hundreds of malware variants with a few clicks. As reported by AV-Test, and seen in Figure 2.1, the number of unique malicious samples released to the public has continued to grow exponentially since 2006 [AV-Test, 2016]. Despite a declining trend in 2016, the amount of new malicious software released to the public still represented the third-highest volume of newly developed malware since AV-TEST Systems first began recording malware data [AV-Test, 2016].

To fight this onslaught of malware, numerous techniques have been proposed over the last three decades. When malicious software first began to appear, the analysis of malware was done manually, which was still a viable option at the time [Landesman, 2017]. Antivirus vendors were successful at detecting and stopping malware attacks by simply identifying every new piece of malware using a unique hash signature (i.e., segments of code that act as unique identifiers for malicious programs) [Ye et al., 2007]. However, signature-based approaches could only recognize previously seen malware variants and failed to detect zero-day malware, i.e., malware for which signatures had not been generated [Pandey and Mehtre, 2014]. As a result, better methods to characterize and classify malware were required. As malware continued to evolve, more sophisticated technologies such as heuristics analyzers (i.e., techniques that use rules to detect malicious files) [Cobb, 2016], as well as dynamic analysis (i.e., methods that

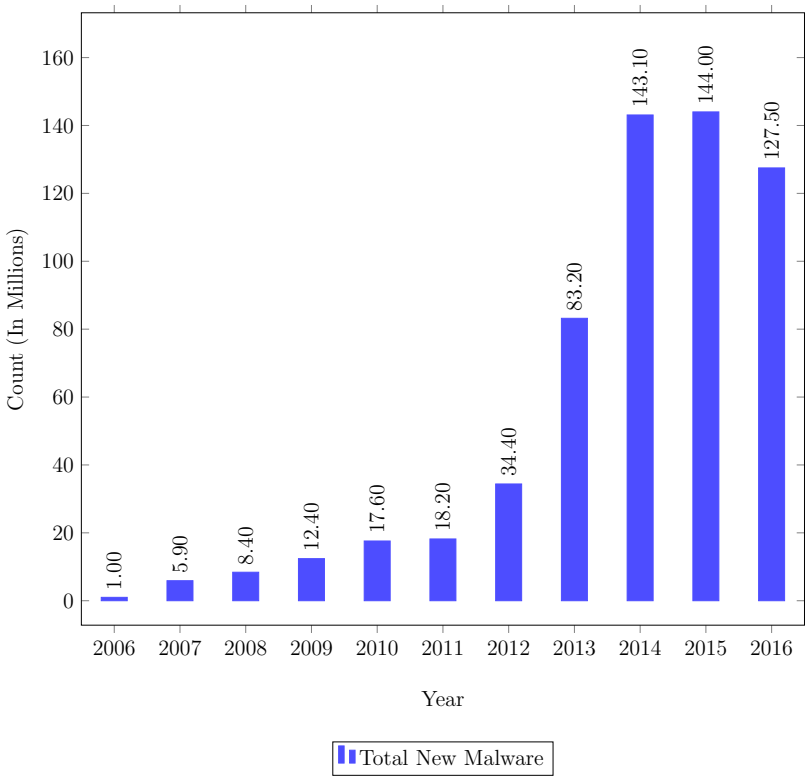

Figure 2.1: This figure shows the exponential growth of unique malware released to the public over the last decade. Despite a declining trend in 2016, the amount of new malicious software released to the public still represented the third-highest volume of newly developed malware since AV-TEST Systems first began recording malware data.

execute and monitor a suspicious file in an emulated environment) [Cuckoo, 2017], and static analysis (i.e., tools that extract a malware's code without executing the program) [Egele et al., 2012], were proposed to overcome the limitations of signature-based approaches [Shevchenko, 2007]. Unfortunately, these methods have weaknesses and often cannot keep up with the massive amounts of new malware variants being released daily.

In the following sections, we introduce and explain the malware analysis techniques that have been employed over the last few decades to analyze and detect malware. Particularly, we focus on static and dynamic analysis, as these methods allow us to extract meaningful characterizations from our malware files and generate features that are used as input to our machine learning models.

2.3 Signature-based Techniques

Signature-based techniques are one of the most common approaches for malware detection. A signature can represent a sequence of bytes that is unique to a program [Ye et al., 2007], as well as a cryptographic hash of a file and its contents [Zeltser, 2001]. Antivirus tools attempt to identify the presence of malware by comparing hashes and bytecode patterns of files on a system against a repository of signatures of known malware [Mujumdar et al., 2013]. When a match is found, the file is flagged as malicious and a mitigation process is initiated. While successful in detecting known malware, signature-based techniques fail to detect zero-day malware [Pandey and Mehtre, 2014], or even polymorphic malware, which are variations of known malware [Tang et al., 2011]. Furthermore, metamorphic obfuscation techniques such as insertion of dead code, register reassignment, and instruction substitution are frequently used by malware authors to change their files' signatures while retaining their malicious functionality [Christodorescu and Jha, 2003].

2.4 Heuristics-based Approaches

In contrast to signature-based approaches, heuristics-based techniques use rules to examine suspicious files and detect malicious intent embedded in the running program without needing a signature [Cobb, 2016]. Malware can be detected by looking for the presence of junk code in the examined file [Yin et al., 2007], modifications to the registry, and insertion of hooks into a specific library or system interfaces [Zeltser, 2001]. However, this approach is limited to the appearance of repeated fragments of code that are indicative of malicious intent [Cade, 2017]. As a result, heuristic rules are inherently brittle, as a small change or mutation to the code of the malware can render them useless. Moreover, malware can evade detection if a proper rule has not been constructed to identify a particular set of malicious actions [Cade, 2017]. Finally, heuristic-based approaches can incur a high number of false positives as they can inadvertently flag a legitimate file as malicious. Since these techniques monitor potentially malicious actions, such as registry modifications, a program displaying a

similar behavior may be deemed malicious despite the application possibly being an important system utility [Yin et al., 2007].

2.5 Static Analysis

Static analysis corresponds to one of the first approaches for malware analysis and detection. It was first introduced by Lo et al. in 1995 [Lo et al., 1995] and is done by extracting a malware's code without executing the actual program [Egele et al., 2012]. Automated tools, such as portable reverse engineering frameworks, can help extract static features from a malware binary in a very accurate and effective manner, providing valuable insights about the intent of the suspicious file [Oktavianto and Muhardianto, 2013]. Other tools, such as call graphs, provide a security analyst with a general overview of the functions that might be called from specific parts in the code of the malware [Shanks, 2014]. In contrast to dynamic analysis, which can take minutes to hours to perform, static analysis can typically be performed in seconds [LMSecurity, 2017]. Another advantage of static techniques is their unique ability to characterize all possible execution paths of a malicious sample while dynamic malware analysis is limited to a single sequence of system calls executed by the program [Nath and Mehtre, 2014]. Finally, static techniques benefit from the fact that a significant portion of newly generated malware corresponds to file-level mutations of previously seen samples [Bayer et al., 2010]. As a result, a full dynamic analysis run can be avoided if the malware detection system identifies that an analysis or a report for a similar program already exists.

2.5.1 Drawbacks of Static Analysis

Static analysis alone is not enough to detect or classify malicious code, because of the ability of certain malware to bypass static methods. Historically, malware has used techniques such as self-modifying code to hamper static analysis and hide its malicious nature [Egele et al., 2012]. In the beginning, malware authors wrote malware in a way that its malicious code was difficult to read without the malware being executed

first. Obfuscation techniques such as polymorphism, in which the payload algorithm of the malware is kept constant while the viral code is mutated [Narayanan et al., 2013], and metamorphism, where a malicious file automatically recodes itself each time it propagates [Hosmer, 2008], were designed by bad actors to avoid detection and thwart static malware analysis [Pirscoveanu et al., 2015]. More recently, malware programs have made use of packers (i.e., programs that transform a malware binary into another form, without affecting its execution semantics, to create new malware variants) and encryption to hamper the disassembly process of malicious code [Guo et al., 2008]. In fact, according to Yan et al. [Yan et al., 2008], a significant portion of malware today comes in packed form, which encumbers detection even more. Furthermore, the application of static analysis is frequently limited by the fact that most of the source code of malicious samples is not readily available [Oktavianto and Muhardianto, 2013]. Therefore, the static analysis of malware poses complex challenges. For example, disassembly can result in incomplete results when the suspicious file uses obfuscation.

2.5.2 Static Methods for Binary Characterization

There are different static methods to characterize binaries as fixed-size features. In our experiments, we leverage three main categories of features: 1) byte-level representations of malware binaries [Saxe and Berlin, 2015], 2) hashes histograms [InfoSec, 2018], [Saxe and Berlin, 2015], and 3) disassembled instructions provided by Radare2 [Searles et al., 2017], [Radare2, 2018]. These sets of features are fed into our machine learning algorithms to induce classification models, as described in subsequent chapters.

2.5.2.1 Bytes Analysis

Byte-level representations of malicious code corresponds to a traditional approach for malware analysis where each file is parsed as a stream of bytes. For our work, we consider two types of byte features:

1. **Bytes Histogram:** represents an aggregated count of each byte value in the file [Saxe and Berlin, 2015].

2. **Bytes-Entropy Histogram:** entropy corresponds to a powerful metric to identify the type of information contained in a file, and measures the randomness in the bytes of the malware [Arora, 2016]. This provides valuable information about malicious files as a high entropy value is usually indicative of the malware being encrypted [Arora, 2016]. Finally, combining the entropy value with the bytes analysis creates additional context capable of more complex learning [Saxe and Berlin, 2015].

Byte-level analysis considers the raw bytes in malicious binaries. This analysis can be used to generate bytes-entropy histograms to identify the amount of information associated with various bytes distributions in our malicious files [Saxe and Berlin, 2015]. One can scan binaries using a sliding window to extract the bytes histogram for each window and compute its entropy. These histograms can then be converted into a two-dimensional bytes-entropy histogram that contains all the byte-entropy pairs for all windows [Saxe and Berlin, 2015]. The process to generate bytes-entropy histograms is depicted in Figure 2.2.

2.5.2.2 Hashes Histograms

Static analysis features can also be constructed from the list of strings included in the file, as well as from the metadata and import table of the Portable Executable (PE) header found in the code of the malware. These features can be transformed into a feature vector of fixed length.

1. **Strings:** correspond to sequences of printable characters. Strings usually provide insights about the functionality or capabilities of malware. For example, strings such as "exec" and "sleep" are normally used to control a remote file, and a field such as "InternetOpenURL" indicates that the malware will try to establish a remote connection to an external server [InfoSec, 2018]. In our experiments, we extract the ASCII strings using the GNU tool *string*.

2. **Portable Executable (PE) header:** the PE format is a data structure that encapsulates the information necessary for the Windows OS to load the executable code [InfoSec, 2018]. Additionally, the PE format contains the following sections:

   (a) **Import Table:** a lookup table that is executed when the program calls a function in a different module [Microsoft, 2018].

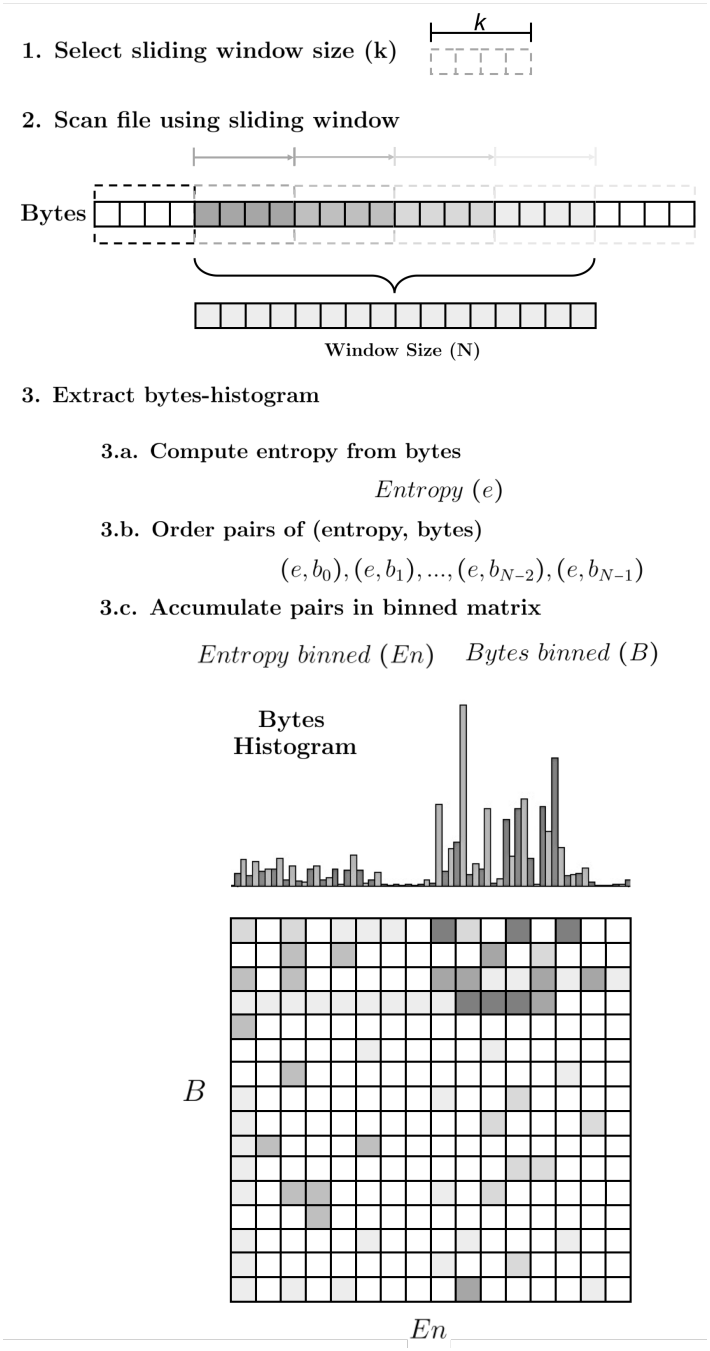

Figure 2.2: This figure illustrates the byte-level characterization of malicious software. We scan malicious files using a sliding window of length 1024 with a step size of 256 bytes. We then proceed to extract the bytes histogram for each window and compute its corresponding entropy. Finally, these histograms are converted into a two-dimensional bytes-entropy histogram that contains all the byte-entropy pairs for all windows.

(b) **Metadata Table:** a table that contains information about different elements of the application, such as the classes, fields, constants, and references among them [Microsoft, 2010].

Each string found in the file can be "hashed" to an integer between zero and the selected vector length, and a histogram of these hashes can be produced by counting the occurrences of each value. The result is a vector of positive integers of the desired size. Figure 2.3 shows how a list of strings is transformed into a fixed-size feature vector.

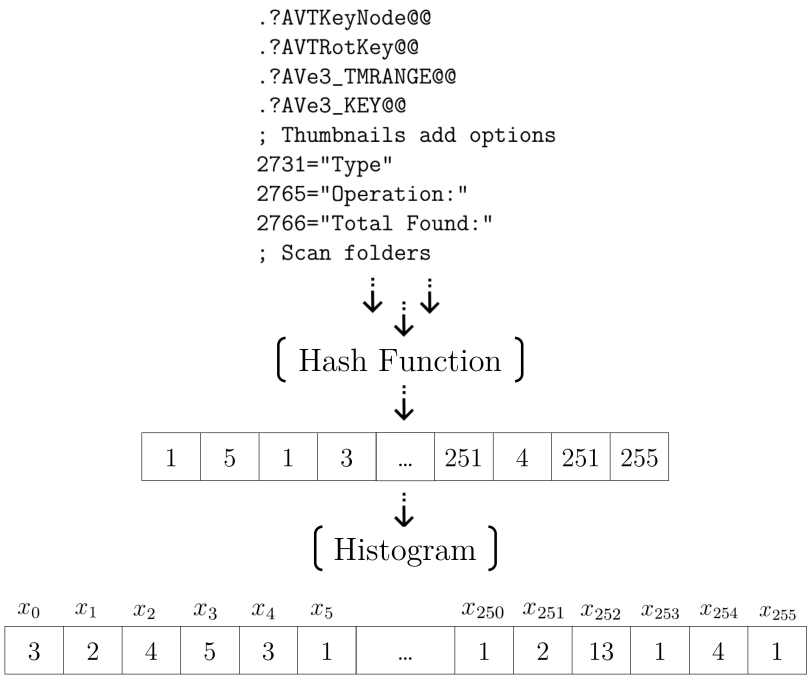

Figure 2.3: This figure shows how features are generated from a malware file's strings. First, each string is "hashed" to an integer between zero and the desired vector length. Second, a histogram of these hashes is produced by counting the occurrences of each value. The result is a vector of positive integers of the desired size.

2.5.2.3 Disassembly Analysis

We are currently using Radare2 to translate the machine code into assembly code (i.e., a human readable representation of machine code [Hope, 2017]). Radare2

corresponds to an *open-source* reverse engineering compiler that was released in February of 2006 [Radare2, 2018]. It can disassemble the executable code of binaries that run on different architectures, including x86, ARM, Bytecode (Java), and Javascript (from HTML files) [Searles et al., 2017]. Instead of considering the raw binaries, Radare2 considers the binary file in term of instructions [Cavazos, 2017]. From the assembly code, we construct different types of compiler representation graphs of the binary, such as Control-flow graphs, from which features are extracted to train machine learning models. In addition, Radare2 provides security analysts with a visual and web user interface that allows them to obtain a more interactive view of the malware binaries [Alvarez, 2016]. Finally, it provides a free and simple command-line hexadecimal editor, with support for 64-bit offsets, to statically analyze malicious executables [Radare2, 2018].

The process to perform disassembly analysis of malware using Radare2 is depicted in Figure 2.4.

First, Radare2 disassembles the executable file into instructions, blocks, and functions, which we define below.

1. **Instructions:** the program statements of the executable file.
2. **Blocks:** sequences of instructions where an operation in each position is executed before the instructions in later positions [Kruegel et al., 2004].
3. **Functions:** a list of blocks that can be analyzed independently and sequentially during the disassembly process [Kruegel et al., 2004].

We then explore the expressiveness of compiler representation graphs constructed from the assembly code by generating three types of graphs: *call-flow, control-flow,* and *data-flow.*

1. **Call-flow graph:** indicates the calling relationships between subroutines in a computer program (i.e., caller-callee relationship between functions) [Searles et al., 2017].
2. **Control-flow graph:** is a directed graph where nodes represent basic blocks, and edges indicate possible control flows from one basic block to another [Hubicka, 2003].

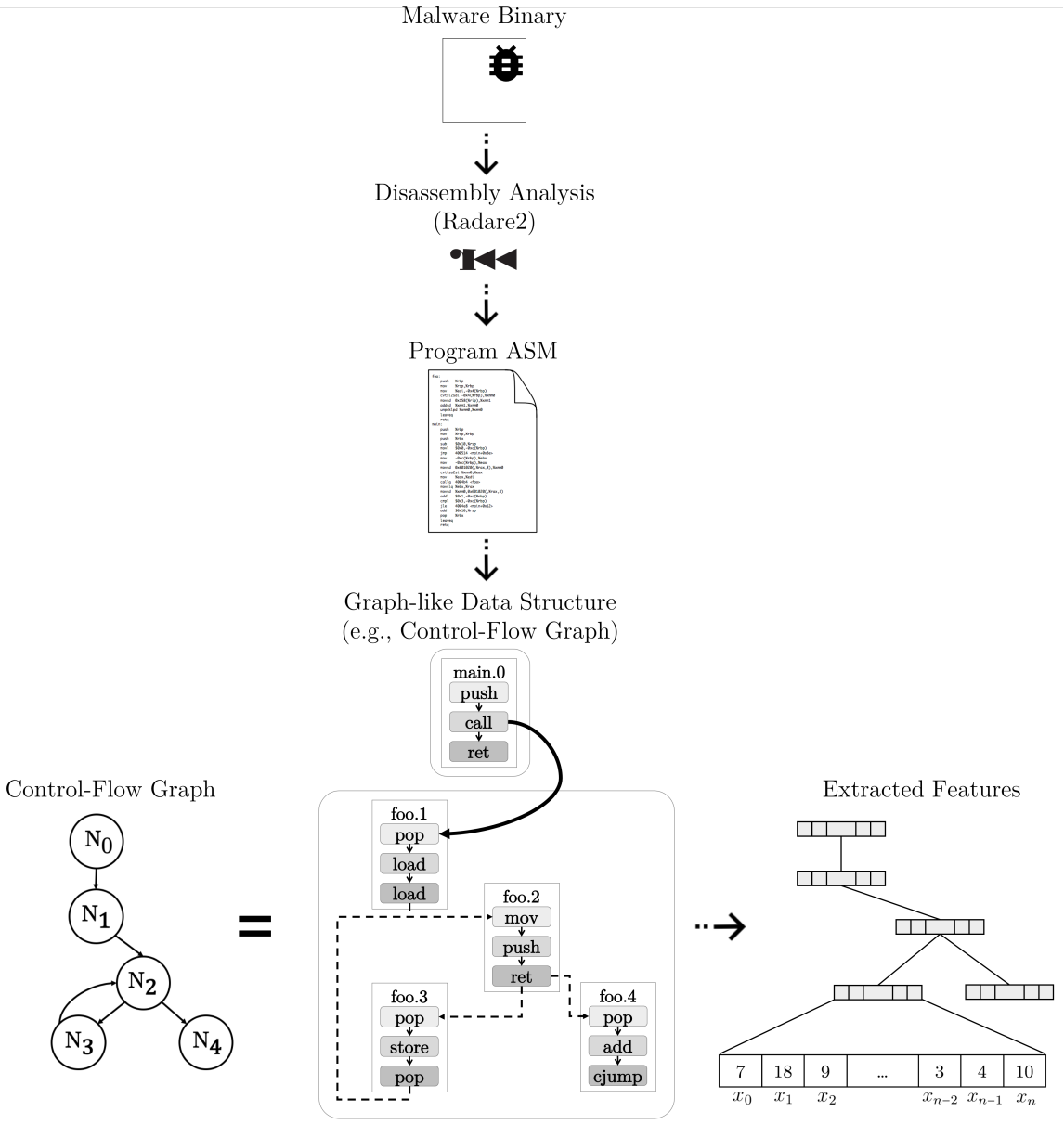

Figure 2.4: This figure shows the process of extracting features from a binary representation using Radare2. First, grouped assembly instructions (ASM) are generated. Secondly, compiler graphs, such as control-flow graphs, are constructed from the disassembled code. Finally, the compiler graphs are regularized to form feature graphs.

3. **Data-flow graph:** represents the read/write dependencies between instructions [Stuart, 2005].

As depicted in the lower middle part of Figure 2.4, compiler representation graphs are composed of operations linked by different types of edges, such as *calls* (black thick curved arrow), *branches* (dotted arrows), and *fall-throughs* (small thin straight arrows). *Edges* between blocks represent jumps in the control flow from the end of a basic block to the start of another [Kruegel et al., 2004]. *Branches* connect the last instruction of one block to the first instruction of the following basic block [Hubicka, 2003]. Branches also implement control structures (e.g., *if* and *switch*) and loops (e.g., *for* and *while*) [Zelenski and Parlante, 2008]. Finally, *fall-throughs* correspond to the edges that indicate the normal flow of instructions (from one basic block to another) in the absence of branching [Hubicka, 2003].

2.5.2.4 Graph-based Features

In order to train our machine learning models, we extract relevant features from the compiler representations of malicious binaries. For each of these graphs, the nodes are labeled with feature vectors representative of the information in the node, where each node represents a block in the original code. Finally, we scan the graph-data structure and accumulate statistics through four levels of granularity: instructions, blocks, functions, and global level [Vanderbruggen and Cavazos, 2017].

1. **Instruction Level:** for each instruction in the whole-program instruction-flow-graph (WPIFG), we aggregate *statistics* that include instruction type and size.
2. **Block Level:** we aggregate statistics, instruction 1-grams, and instruction 2-grams for each block of the whole-program control-flow-graph (WPCFG). Statistics contain the block's size, edges' statistics, and aggregated averages of the block's operations. *Instruction 1-grams* and *2-grams* correspond to histograms of sequences of one or two instructions.
3. **Function Level:** for each function of the call-graph (CG), we aggregate instructions 1-grams, instruction 2-grams, and statistics, that include information about the function and aggregated averages of its blocks and operations.
4. **Global Level:** we collect statistics about the code size, number of operations, blocks, functions, and their corresponding aggregated averages.

2.6 Dynamic Analysis

Dynamic analysis (or behavioral analysis) is a process in which a malicious sample is executed and its behavior, as well as the changes that occurred in the analysis system, are monitored and reported for further examination [Cuckoo, 2017]. Reports derived from dynamically analyzed malware usually include the following results:

1. **File activities** highlight the archives that were created or deleted during the execution of the suspected sample [Bayer et al., 2010].
2. **Traces of system calls** denote requests spawned by the malware to perform an operation or run a program on the analysis system [Thakur, 2016].
3. **Windows registry activities** describe modifications of the information and configuration settings of the software and hardware installed on a Windows operating system [Fisher, 2017].
4. **Memory dumps** correspond to the analysis of the contents of the system's Random Access Memory (RAM) to extract information such as running processes, network connections, and modules loaded by the malware [Ka, 2016].
5. **Network activities** indicate the files that were downloaded, the information sent over the network, and malicious websites contacted by the malware [Vadrevu and Perdisci, 2016].

Furthermore, dynamic malware analysis has emerged as a state-of-the-art detection approach that compensates for the shortcomings of static analysis techniques [Hu et al., 2013]. Schemes such as obfuscation and run-time packing are less likely to affect the dynamic execution of a suspicious file [Hu et al., 2013].

Nevertheless, given that the execution of a malicious sample can lead to undesired consequences (e.g., file deletion and modification, loss of confidential information, and changes in registry), dynamic analysis must be performed in a safe environment or *sandbox*, which is a confined execution system that can be used to isolate and run unreliable software and observe its malicious behavior [Oktavianto and Muhardianto, 2013]. By a using a sandbox, it is possible to perform dynamic analysis without worrying about the changes that will occur during the execution of the suspicious sample.

In our experiments, we use Cuckoo Sandbox to dynamically analyze potentially malicious applications in an isolated and safe environment. This allows us to extract

meaningful information and generate dynamic features, such as the malware analysis run time and threat level assigned to a suspicious sample, both of which are used as input to our machine learning algorithms.

2.6.1 Drawbacks of Dynamic Analysis

While dynamic analysis can examine malware that is resistant to static analysis, it is still limited by the ability of some malware to detect a sandbox environment and stop the execution of any malicious activity [Kirat et al., 2014]. In addition, dynamic analysis often requires a significant amount of time (e.g., five minutes) to execute malicious samples, which poses a challenge as quickly performing dynamic analysis is essential for coping with the ever-increasing load of malware that appears every day [Bayer et al., 2010]. Finally, dynamic malware analysis has limited coverage of the behavior of the malware as it only captures the behavior (e.g., API calls, registry changes, and network activities) that corresponds to the execution path taken during a particular run of the sample [Hu et al., 2013]. Moreover, bad actors also add logic to their malware to hide its malicious nature and to perform certain operations only under specific conditions (e.g., a particular date/time is reached or input from a user is detected in the analysis environment). These triggered-based malware generate fewer run-time traces, which results in an incomplete picture of their malicious activity when dynamic analysis is performed [Hu and Shin, 2013].

2.6.2 Cuckoo Sandbox

Cuckoo is an open source sandbox for malware analysis [Cuckoo, 2017]. It is used to automatically analyze suspicious samples and record their behavior while running inside an isolated environment. Cuckoo began as a Google's Summer of Code project in 2010, and its first beta version was announced and released in 2011 [Oktavianto and Muhardianto, 2013].

In January 2012, Cuckoo launched malwr.com, a free and public environment for malware analysis [Oktavianto and Muhardianto, 2013]. Security analysts can use

this instance to submit files for dynamic analysis and obtain a report outlining their behavior. The most recent release of Cuckoo (version 2.0) was officially announced in 2017 [Cuckoo, 2017]. This version simplified the usage and maintenance of Cuckoo, making it more accessible for analysts interested in performing dynamic analysis of malware.

Furthermore, Cuckoo offers the following benefits over other traditional sandbox environments [Cuckoo, 2017]. It lists the files that were created, deleted, and downloaded by the malicious sample during its execution [Cuckoo, 2017]. Additionally, it dumps and analyzes network activities and traces system calls, including services that were installed and processes that were terminated during the analysis of the sample [Oktavianto and Muhardianto, 2013]. Moreover, Cuckoo can inspect and record the behavior of any file, including Windows executables, Portable Document Format (PDF) files, Java applets, Microsoft Word documents, mobile phone apps, and malicious websites [Shanks, 2014].

In addition, Cuckoo has the following advantages over other commercial approaches for dynamic analysis such as JoeSandbox [JoeSandbox, 2017] and ThreatExpert [ThreatExpert, 2017]. It is fully open source and has a modular design that enables the customization of the analysis environment. It also is supported by a strong and active community of developers and security analysts. Finally, Cuckoo can easily be integrated into an existing framework for malware analysis without licensing requirements [Cuckoo, 2017].

2.6.2.1 Architecture

The general architecture of Cuckoo is depicted in Figure 2.5 and consists of two main components [Oktavianto and Muhardianto, 2013]:

1. **Host machine:** is the underlying operating system on which we deploy Cuckoo (usually a GNU/Linux Distribution such as the Ubuntu operating system) [Cuckoo, 2017]. It manages the core functions of the sandbox, such as submitting files for analysis to the isolated environment, and generating reports about the behavior of the samples.

2. **Guest machines:** are the safe and confined environments that run and analyze malware and report its behavior back to the host [Oktavianto and Muhardianto, 2013].

3. **Malware initiator:** a critical component of a malware analysis system which starts the execution of the binary [Kirat et al., 2014]. In our case, the initiator takes the form of an in-guest agent. This cross-platform initiator receives the malicious sample from the host machine through a local network, starts the execution of the file, and communicates the analysis results back to the host. Furthermore, a start-up entry is generated in the system to ensure that the agent is running after each system reboot and to control the duration of the execution of the malware.

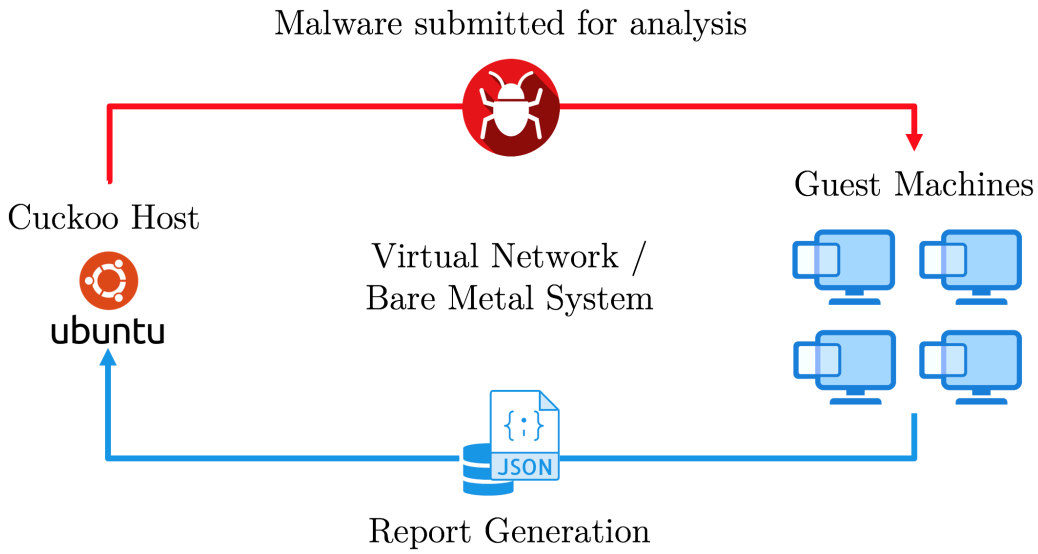

Figure 2.5: This figure shows Cuckoo's architecture. The host manages the main functions of the sandbox, such as submitting files for analysis and generating reports about the behavior of the samples. The guests correspond to isolated environments that safely execute and analyze malware, and report its behavior back to the host. These guest machines are part of a virtual network or bare-metal architecture (i.e., a system running on real hardware).

2.6.2.2 Processing Modules

Cuckoo's modular framework allows users to define custom ways to process the results derived from the execution of the malware. The core processing modules of Cuckoo are listed and defined below [Oktavianto and Muhardianto, 2013].

1. **AnalysisInfo** provides some basic-level information about the analysis, such as the timestamps for the system calls executed by the malware [Oktavianto and Muhardianto, 2013].

2. **BehaviorAnalysis** processes the raw dynamic logs to generate a behavioral summary of the malicious actions performed by the malware, as well as a process tree that outlines the relationships between these actions [Oktavianto and Muhardianto, 2013].

3. **Dropped** describes the files downloaded by the malware during its execution [Oktavianto and Muhardianto, 2013].

4. **NetworkAnalysis** extracts network information such as IP addresses and domains the malware tried to contact [Oktavianto and Muhardianto, 2013].

5. **TargetInfo** presents information such as the hash of the analyzed file (i.e., the cryptographic fingerprint that uniquely identifies malware [Honig and Sikorski, 2012]).

6. **VirusTotal** looks up the hash of the file on VirusTotal.com for antivirus signatures [Oktavianto and Muhardianto, 2013].

7. **ProcMon** provides information regarding the file systems, registries, and other activities triggered during the analysis of the malware [Zeltser, 2011].

8. **Droidmon** includes information about dynamic API calls spawned by the malware [Oktavianto and Muhardianto, 2013].

2.6.2.3 Repository of Malicious Behaviors

The developers of Cuckoo have a-priori determined a set of behaviors that could be deemed as malicious, such as a file installing itself for autorun during Windows startup or querying for a computer name information [Cuckoo, 2017]. When a malicious sample is executed in the sandbox environment, it is possible to run these behaviors against the analysis results and identify patterns or indicators of interest [Oktavianto and Muhardianto, 2013]. These behaviors provide a context to the analyses as they simplify the interpretation of the results and can help identify particular malware families (e.g., ransomware) by isolating unique malicious actions performed by these types of malware [Oktavianto and Muhardianto, 2013]. For our work, we explore this behavioral space and create models that predict the important capabilities of the

samples in our malware dataset (e.g., modification of windows registries, creation and deletion of files and network activities, among others).

2.6.2.4 Maliciousness Scale

After a suspicious sample is dynamically analyzed in the sandbox environment, Cuckoo provides a measurement that aims to quantify the maliciousness of the file [Cuckoo, 2017]. The aforementioned malicious behaviors are given a severity score, and the scores are combined into a single maliciousness scale from 1 to 10. Finally, Cuckoo classifies a file as malicious if the final score is equal or greater than 5.0 [Oktavianto and Muhardianto, 2013].

2.7 Machine Learning for Cyber Security

In this section, we introduce several studies that are most relevant to the work described in this dissertation. We present a summary of existing research that uses machine learning, in conjunction with features extracted from static and dynamic analysis, to detect and classify malicious malicious executables.

2.7.1 Using Static Analysis Features

Static features extracted from malware such as PE headers [Wicherski, 2009], instruction sequences [Karim et al., 2005], binary sequences [Jang et al., 2011], and code patterns [Christodorescu and Jha, 2003] have been largely used to train machine learning algorithms and build predictive models for malware detection and categorization of zero-day malware [Shafiq et al., 2009].

The work of Shultz et al. [Schultz et al., 2001] is widely known for first introducing the concept of machine learning for malware detection. In their research, the authors leveraged naive Bayes classifiers along with static analysis features to detect malware. The static features selected to train their learning algorithm corresponded to byte-sequence n-grams, portable executables (PE) headers, and string features. Their results showed that a naive Bayes algorithm trained on strings extracted from malicious

code could yield a classification accuracy of up to 97.11%, outperforming traditional methods such as signature-based techniques [Schultz et al., 2001].

The approach proposed by Schultz et al. was later improved by Kolter and Maloof [Kolter and Maloof, 2004]. They used n-grams, as opposed to byte sequences, to train a set of machine learning algorithms such as decision trees, naive Bayes, and support vector machine, achieving the best classification rate with the boosted version of the decision tree algorithm [Kolter and Maloof, 2004].

Tien et al. [Tian et al., 2008] leveraged function length frequency, i.e., a measure of the number of bytes in the code of malware, coupled with machine learning algorithms provided by the Weka library [Hall et al., 2009], to effectively classify malicious executables [Tian et al., 2008].

Similarly, Siddiqui et al. [Siddiqui et al., 2009] employed decision trees and random forest models to classify malware based on variable length instruction sequences [Siddiqui et al., 2009]. The work of Schmidt et al. [Schmidt et al., 2009a], [Schmidt et al., 2009b] demonstrated the high detection rates that could be achieved by incorporating the function calls extracted from static analysis with machine learning techniques (support vector machine and naive Bayes) to classify malware [Schmidt et al., 2009a], [Schmidt et al., 2009b].

Other approaches based on more complex features extracted from malicious code have also been proposed for malware detection and classification. Kong and Yan [Kong and Yan, 2013] introduced a framework for malware classification based on features extracted from the function call graph of malware. Using similarity learning and an ensemble of classifiers, the authors were able to classify malicious samples in their corresponding families with an accuracy of 93.33% [Kong and Yan, 2013].

Furthermore, more intricate approaches have been proposed to detect and classify malware based on static-based features. Dahl et al. [Dahl et al., 2013] extracted strings and API tri-grams from malicious code and used this feature space to train multiple neural network architectures, achieving an error rate of 0.49% for the classification of malware [Dahl et al., 2013].

Saxe and Berlin [Saxe and Berlin, 2015] developed a deep learning architecture trained on ASCII strings and bytes-entropy histograms using feedforward neural networks to detect malware. Their platform managed to yield a detection rate of almost 95% based on more than 400,000 malware binaries [Saxe and Berlin, 2015]. To the best of our knowledge, this work represents the most effective tool for detecting malware based solely on static characterizations.

2.7.2 Using Dynamic Analysis Features

Other researchers opt to work with dynamic analysis techniques to improve the effectiveness and efficiency of malware detection and classification. For example, API calls have been largely used to train machine learning classifiers to identify and characterize malware [Anderson et al., 2011], [Bayer et al., 2009], [Tian et al., 2010], [Bailey et al., 2007], [Park et al., 2010], [Firdausi et al., 2010]. They indicate the requests spawned by the malware to perform an operation or run a program on the analysis system [Thakur, 2016].

Dynamic analysis has gained in popularity due to the fact that schemes that normally thwart static analysis, such as obfuscation and run-time packing, are less likely to affect the dynamic execution of a suspicious file [Hu et al., 2013]. In the following paragraphs, we describe some of the approaches developed to train machine learning models based on this particular dynamic characterization of malware.

Tian et al. [Tian et al., 2010] examined API call sequences extracted from dynamic analysis of malicious executables. Using classifiers available in the WEKA library such as decision tables, random forest, and support vector machine, they were able to differentiate between malware and goodware and classify malicious samples into their corresponding families, with an overall accuracy of up to 97% for both problems [Tian et al., 2010].

Firdausi et al. [Firdausi et al., 2010] leveraged dynamic reports obtained from the execution of malware in a sandbox environment. Information about the API calls spawned by the malicious executables is transformed into sparse feature vectors for

classification of malware using classifiers such as support vector machine, k-nearest neighbors, naive Bayes, and decision trees. The best performing model corresponded to the J48 decision tree algorithm, which managed to accurately classify malware with an average accuracy of 96.8% [Firdausi et al., 2010].

Furthermore, more complex approaches, such as neural networks in the context of supervised learning, have proved to be well-suited for malware classification. Kolosnjaji et al. [Kolosnjaji et al., 2016b] trained a neural network with convolutional and recurrent network layers using system call sequences extracted from dynamic analysis reports. Their results outperformed traditional machine learning methods for malware classification, such as hidden Markov models and support vector machine, achieving an average precision and recall of 85.6% and 89.4%, respectively [Kolosnjaji et al., 2016b].

Huang and Strokes [Huang and Stokes, 2016] proposed a deep neural network architecture that employed API call events and null-terminated strings (i.e., an object where the character "\0" signals the end of the string) as input feature vectors to their deep learning framework. Their approach was able to yield an error rate of 2.94% for the problem of malware classification, and 0.358% for the task of detecting malware [Huang and Stokes, 2016].

Hardy et al. [Hardy et al., 2016] constructed an intelligent framework for malware detection based on deep learning. Their approach leveraged the stacked AutoEncoders (SAEs) model and features constructed from sequences of API calls to identify malicious binaries. With an estimated accuracy of over 96.85%, their platform was able to exceed the results obtained by other shallow learning-based classification methods such as support vector machine, naive Bayes, and decision tree [Hardy et al., 2016].

Pascanu et al. [Pascanu et al., 2015] attempted to learn the "language of malware" by examining semantic representations of instructions executed by malicious files in an emulated environment. In their research, the authors used recurrent neural networks and echo-state networks to learn from event sequences and effectively detect reordered malware [Pascanu et al., 2015].

In addition, features such as network activities have also been used to induce

highly accurate malware detection and classification models. Nari and Ghorbani [Nari and Ghorbani, 2013] developed a framework that leveraged network behavior to perform malware family classification. After constructing behavioral graphs from network traces exhibited by the malware, features such as graph size and number of nodes were extracted to perform malware classification using the the decision tree algorithm. Their results surpassed the performance obtained by anti-virus programs in classifying malware, achieving an overall accuracy of nearly 95% [Nari and Ghorbani, 2013].

Finally, Rieck et al. [Rieck et al., 2011] proposed a framework for the automatic analysis of malware behavior using machine learning. In their work, the authors used an incremental approach for behavior-based analysis, which was capable of processing the behavior of thousands of malware binaries on a daily basis. Their framework automatically identifies novel classes of malware with similar behavior and assigns unknown malware to these discovered classes [Rieck et al., 2011].

2.7.3 Using Hybrid Features

In addition to static and dynamic analysis, there are a few documented cases of research that combines these features into a hybrid approach to detect and classify malware.

Santos et al. [Santos et al., 2013] used opcode sequences extracted from static analysis and a large number of dynamic features to build a malware detection framework. Their platform employed supervised learning (specifically, decision tree, k-nearest neighbor, Bayesian networks and support vector machine) to classify unseen malware with high accuracy [Santos et al., 2013].

Islam et al. [Islam et al., 2013] employed function length frequency and string information from static analysis, as well as API calls from dynamic analysis to train support vector machine, decision tree, random forest and instance-based classifiers. This improves on previous results based on individual features and significantly reduces the time spent testing features from static and dynamic analysis separately [Islam et al., 2013].

Anderson et al. [Anderson et al., 2012] leveraged features extracted from static analysis (e.g., control flow graph) and dynamic analysis (e.g., dynamic instruction trace and system call trace) to train a support vector machine classifier to detect malware [Anderson et al., 2012].

Finally, a more recent paper by Kolosnjaji et al. [Kolosnjaji et al., 2016a], showed how supervised learning applied to a hybrid model that combines features from static and dynamic analysis significantly displays more robustness than traditional approaches and increases the accuracy of inferred class labels [Kolosnjaji et al., 2016a].

2.8 Discussion

Based on the previous summary, it is evident that there is a deep relationship between static and dynamic malware analysis and machine learning, as classifiers can aid in categorizing malware for which signatures have not been recorded. By taking advantage of both static and dynamic features, it is possible to construct models that extract the underlying structure of the data of a malicious file and help researchers make predictions from these analyses. Nevertheless, the aforementioned methods have significant weaknesses that we highlight in the following paragraphs.

Models trained exclusively on static analysis features are limited due to the vulnerability of this approach to obfuscation techniques, such as insertion of dead code, register reassignment, and instruction substitution [Christodorescu and Jha, 2003]. In fact, Grosse et al. [Grosse et al., 2016] demonstrated that the accuracy of a machine learning classifier can dramatically decrease when trained on adversarial malware that incorporates obfuscation.

Although classifiers built on dynamic-based features overcome some of the limitations of models trained using static analysis information, they still suffer from performance and scalability issues [Bierma et al., 2014]. Existing approaches that leverage machine learning and dynamic data to detect and classify malware such as [Huang and Stokes, 2016] and [Hardy et al., 2016] use the entire dynamic execution cycle of suspicious samples. Malware takes a long time to be executed in emulated environments,

which creates an impractical delay (in comparison with static techniques) if dynamic analysis is used as part of an endpoint security system [Rhode et al., 2018]. As a result, dynamic analysis cannot efficiently scale to large malware datasets consisting of millions of samples.

In the following chapters, we prove that the malicious nature of most malware can be detected in a short amount of time, avoiding a full dynamic analysis on the files. By leveraging static analysis features extracted from unpacked malware, i.e., malicious binaries from which all packing, obfuscation, and protection artifacts have been removed, we construct models that estimate and reduce the malware execution time and increase the throughput of our malware detection platform. In addition, we also demonstrate that dynamic analysis is not always required to detect and classify malware and that simpler characterizations are often enough to make a correct prediction. Finally, using features extracted from fast static characterization of unpacked malware, we are able to induce accurate models for malware capability detection, removing the need to perform dynamic analysis to extract high-level functionalities of malicious code.

Chapter 3

PROPOSED APPROACH

In this dissertation, we propose an intelligent malware analysis system that leverages machine learning to detect and classify new malware in a short amount of time and infer its important capabilities. Our malware analysis system can identify the simplest features required to accurately classify malware. This allows us to determine the subset of samples that is truly different and requires very expensive dynamic characterization. When dynamic analysis is imperative, our system also estimates the minimum amount of time required to detect and classify malware. This is a crucial quality because the traditional process of dynamic analysis can be a bottleneck for studying malware [Damodaran et al., 2017]. It is therefore essential that our malware analysis system spends the least amount of time analyzing malware whose malicious nature can be detected in seconds. The time savings can then be reallocated to analyzing files that require longer execution times and produce actionable intelligence for the system [Vadrevu and Perdisci, 2016]. Finally, we leverage the speed of static analysis [LMSecurity, 2017] to induce highly accurate machine learning models for malware capability detection.

Our approach determines the minimum amount of time needed to detect the malicious nature of malware. In contrast, traditional sandbox environments execute suspected samples for long periods of time (often five minutes or more [Bierma et al., 2014], [Quist et al., 2011], [Tobiyama et al., 2016]), regardless of the information extracted from the file in question [Bayer et al., 2010].

In addition, our approach uses machine learning to automatically deduce important capabilities of malware. In contrast, traditional approaches for describing malware, such as YARA, depend entirely on a large number of rules that are inherently brittle

and difficult to maintain [Saxe et al., 2013]. YARA is an open-source project that tries to predict important capabilities of malware [Alvarez, 2017]. YARA provides a framework to generate descriptions of malware families based on textual or binary patterns. Nevertheless, a small change or mutation to the code of the malware can render these patterns useless [Cade, 2017]. Our approach, on the other hand, leverages automated machine learning-based models that infer high-level functionalities of malicious code.

Furthermore, our system uses fast static analysis to predict important capabilities of malware. Other automated techniques rely on slow dynamic analysis to create behavioral profiles that describe high-level functionalities of malicious code [Yavvari et al., 2012].

Moreover, our system creates descriptions of malware capabilities based on the decision tree algorithm. This technique offers unique advantages since its output has the potential to be human interpretable and easy to visualize [Petri, 2015]. Decision trees also inherently perform useful tasks, including feature importance and selection on a dataset [Deshpande, 2011a]. We construct visual models that provide security analysts with more information about the specific elements of malicious code associated with a particular capability. Other machine learning algorithms, such as deep learning, generate representations that are more difficult for security analysts to interpret [Holmes, 2014]

Additionally, our system can effectively single out the subset of malware with anti-sandbox capabilities that requires execution on a bare-metal (i.e., physical) machine to display its malicious behavior. In contrast, other automated approaches run large malware datasets on bare-metal architectures (in an attempt to mitigate the side-effects of virtualized environments), incurring high hardware costs and suffering from scalability limitations [Kirat et al., 2014].

Finally, our system speeds up feature generation by comparing the predictive power of different malware characterizations, and selecting the least computationally expensive features that lead to a correct classification. As a result, expensive characterizations are not always required. In contrast, other automated techniques select

features that can lead to accurate predictions but demand high computational power to be generated [Ahmadi et al., 2016].

The contribution of this dissertation corresponds to the development of an intelligent sandbox environment that significantly extends the capabilities of Cuckoo, an open-source malware analysis system [Oktavianto and Muhardianto, 2013]. A set of machine learning algorithms are used to build three predictive models and address several important sandbox problems currently being solved by manually-tuned heuristics: 1) predicting the length of time needed to execute a malware, 2) learning when dynamic behavior improves the accuracy of malware detection beyond performing static analysis, and 3) automatically identifying important capabilities of malware.

3.1 TURACO: Training Using Runtime Analysis from Cuckoo Outputs

We demonstrate that the malicious nature of most malware can be detected in a short amount of time (as short as ten seconds), without requiring a full dynamic analysis on the files.

Traditional sandbox environments execute suspected malware for long periods of time [Damodaran et al., 2017]. In addition, the overall run time might be longer in some cases, as numerous systems include a post-processing phase following the analysis of the suspicious sample [Bayer et al., 2010]. This creates an impractical delay if dynamic analysis is used as part of an endpoint security system [Rhode et al., 2018]. Most malware, however, do not require long executions, because a model may be able to predict their malicious intent and corresponding family after observing their behavior for only a few seconds [Rhode et al., 2018].

We introduce an approach that accurately determines the amount of time needed to execute malware in a sandbox environment. Specifically, we develop a technique that we refer to as *TURACO* (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs). This model leverages features extracted from fast static analysis of malware, as well as dynamic reports generated by Cuckoo. By feeding this information into a machine

learning model, TURACO can predict the minimum amount of time needed to reveal the malicious nature of malware with an estimated accuracy of over 93.86%.

3.2 SEEMA: Selecting the Most Efficient and Effective Malware Attributes

Our automated malware analysis system also uses a model called *SEEMA* that **S**elects the most **E**fficient and **E**ffective **M**alware **A**ttributes to assign malicious binaries into their corresponding families. Malware family classification corresponds to the process of determining whether a binary is part of a previously seen family, or if it corresponds to a new unseen sample that requires further examination [AlAhmadi and Martinovic, 2018].

A malware family provides security analysts with crucial information about the threat level that a file poses to the system, as well as the remediation process that needs to be followed to mitigate a potential threat [Kaspersky, 2018]. For example, the incident response for ransomware differs from a botnet's mitigation process, as the former represents a larger threat to a system and must be prioritized accordingly [AlAhmadi and Martinovic, 2018].

Using dynamic analysis for malware classification can be computationally expensive and should not be considered for every sample that is part of a large malware dataset [Kolosnjaji et al., 2016b]. Less expensive characterizations of malware, such as static analysis, can be used to generate descriptive features to be used in conjunction with machine learning to accurately predict malware families [Saxe and Berlin, 2015]. Nevertheless, for some malware it may still be necessary to use dynamic-based features to produce a better prediction [Islam et al., 2013].

We extract various malware characterizations from both static and dynamic analysis and explore different feature sets to feed as input to our machine learning algorithms. We construct classification models based on a set of features of increasing computational cost and predictive power, ranging from cheap-to-compute static-based features to more computationally expensive dynamic-based features. This approach allows us to reduce the overall time it takes to generate features for a significantly

large malware dataset. SEEMA can determine the simplest characterizations required to correctly classify malware, and the subset of samples that truly need more expensive dynamic analysis to be correctly classified. Our results indicate that SEEMA can predict the simplest features and models to classify malware with an accuracy of up to 95.11%.

3.3 MAGIC: Malware Analysis to Generate Important Capabilities

We use our intelligent sandbox environment to induce machine learning models to predict important capabilities of malware. Malware capabilities are high-level functionalities of malicious software, such as the ability to capture key strokes or encrypt a user's personal information [Saxe et al., 2014].

Machine learning models trained using features extracted from the dynamic execution of malicious code can be used to predict behavioral components of malware and infer its important capabilities [Yavvari et al., 2012]. However, dynamic analysis usually limits the throughput of an analysis system, because traditional sandbox environments take a long time to execute suspected malware [Damodaran et al., 2017]. This poses a challenge since dynamic analysis needs to be orders of magnitude faster to cope with the ever-increasing load of malware that appears on a daily basis [Vadrevu and Perdisci, 2016].

Static analysis helps mitigate the computational expense of dynamic analysis, as it can often be performed in seconds [LMSecurity, 2017]. As a result, machine learning models trained on static features can lead to predictive classifiers that can scale to large amounts of malware [Costin et al., 2014].

We use machine learning to construct a model that we refer to as *MAGIC* (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities) that learns from static and dynamic characterization of malware and automate malware capability discovery. We use cheap-to-compute static features of malicious code and capabilities obtained from Reversing Labs (a malware dataset provider) and Cuckoo Sandbox, to map fast static analysis characteristics that indicate malicious behaviors. By using inexpensive static

characterization of our malware dataset, MAGIC can dramatically reduce the time that is required to deduce high-level functionalities of malware. Finally, MAGIC enables us to predict the important capabilities of malware with an accuracy of up to 97.11%.

3.4 Summary of Contributions

We summarize our contributions:

1. We introduce TURACO (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs) to automatically estimate the amount of time needed to execute malware and reveal its malicious intent.

2. We introduce a model called SEEMA (**S**electing the most **E**fficient and **E**ffective **M**alware **A**ttributes) that determines when simple and less expensive characterization of malware, such as static analysis, will suffice to accurately classify malicious executables or when more computationally expensive descriptions, such as dynamic analysis, are required.

3. We introduce a model that we refer to as MAGIC (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities) that learns from static and dynamic characterizations of malware and automatically induces its important capabilities.

Chapter 4

TURACO: TRAINING USING RUNTIME ANALYSIS FROM CUCKOO OUTPUTS

4.1 Introduction

Malware analysis sandboxes are isolation and containment environments where new software can be forced to run prior to allowing it to execute in an enterprise. Sandboxes can inspect the behavior of any file, including Windows executables, PDFs, Microsoft Word documents, mobile phone apps, etc. In this chapter, we refer to malware samples executing in a sandbox without any loss of generality. Sandboxes have become an excellent cybersecurity tool in the security analysts arsenal and have continued to gain in popularity over the last decade. Platforms such as CWSandbox [Willems et al., 2007], JoeSandbox [JoeSandbox, 2017], and ThreatExpert [ThreatExpert, 2017] have been developed to help security analysts monitor the behavior of malicious samples in a confined environment. In these traditional sandbox environments, malware is usually executed for a fixed amount of time regardless of the information extracted from the file in question (e.g., one minute, [Kirat et al., 2011], two minutes [Willems et al., 2007], three minutes [Hansen et al., 2016], four minutes [Christodorescu et al., 2008], or even five minutes [Damodaran et al., 2017], [Tobiyama et al., 2016],[Christodorescu et al., 2008]. However, a significant number of malicious binaries submitted for dynamic analysis are repackaged versions of previously seen and analyzed malware [Vadrevu and Perdisci, 2016]. This results in a large amount of time being wasted in studying files that do not improve the intelligence of the analysis system.

In this chapter, we investigate the application of machine learning to predict the time needed to reveal the malicious intent of a file. We develop a model called *TURACO* (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs) that significantly extends

the capabilities of Cuckoo. TURACO leverages features extracted from *fast* static analysis of malware, as well as reports generated by Cuckoo that provide insights about the time needed to accurately classify a file as malicious. By feeding this information into a machine learning model, TURACO can predict the minimum amount of time needed to deem a file as malware.

4.2 Problem Statement

One of the biggest challenges of dynamic analysis systems corresponds to having to process and provide meaningful reports for an ever-increasing number of malware, within a given period of time and with a limited set of computational resources [Vadrevu and Perdisci, 2016]. Therefore, it is critical to estimate the shortest amount of time required to execute a suspicious file for it to reveal its malicious intent. In traditional sandbox environments, malware is examined for a fixed amount of time, as indicated in Table 4.1 [Rhode et al., 2018], or until the execution of the malware ends [Bayer et al., 2010].

Reference	**Time required for dynamic malware analysis**
[Kirat et al., 2011]	1 minute
[Willems et al., 2007] [Vadrevu and Perdisci, 2016]	2 minutes
[Hansen et al., 2016]	3 minutes
[Christodorescu et al., 2008]	4 minutes
[Bayer et al., 2010] [Bierma et al., 2014] [Anderson et al., 2011] [Quist et al., 2011] [Damodaran et al., 2017] [Tobiyama et al., 2016] [Storlie et al., 2014]	5 minutes

Table 4.1: Reported time for dynamic malware analysis

This time is naturally reduced, should the execution of the malware exit or stop (because of an error) before the timeout is reached. On the other hand, the overall

run time might be longer, because numerous systems allow a post-processing phase following the analysis of the binary sample. For instance, certain systems allow for an additional step of four minutes, on top of the actual execution of the malware [Bayer et al., 2010].

4.2.1 Dynamic Analysis Time

Bayer et al. [Bayer et al., 2010] formally define the overall dynamic analysis time as follows:

$$OverallDynamicAnalysisTime = (|B| \cdot \sum_{b \epsilon B} t_a(b))/I \tag{4.1}$$

with B being the set of binaries, t_a the analysis time for a single malware binary, and I the total number of instances of the analysis system that are running in parallel [Bayer et al., 2010]. Specifically, the analysis time $t_a(b)$ of a binary b is composed of a *setup time* $t_s(b)$ and a *post – processing time* $t_p(b)$, in addition to the time $t_e(b)$ used for executing the binary in a secure environment, as indicated in Equation 4.2 [Bayer et al., 2010].

$$t_a(b) = t_s(b) + t_e(b) + t_p(b) \tag{4.2}$$

During the setup time, the analysis environment is prepared (possibly by loading a virtual machine and transferring the program into it). In the execution time, an initiator starts the analysis of the malware. This initiator can take the form of an in-guest agent which receives the malware from the host machine through a local network [Kirat et al., 2014]. In the post-processing step, various kinds of offline analysis methods are applied to the information gained during the execution of the binary [Oktavianto and Muhardianto, 2013]. Tasks performed during this stage range from archiving the analysis results and updating databases, to running scripts for analyzing a network traffic dump file [Oktavianto and Muhardianto, 2013].

As mentioned earlier, numerous malware include logic to circumvent dynamic analysis by performing their malicious actions after a long period of time [Assor, 2016].

Consequently, longer executions might be required in some cases to reveal the entire capabilities of a sample, exacerbating the resource-intensive nature of dynamic malware analysis [Bayer et al., 2010].

Given the increasing stream of new malware that is generated every day [Harrison and Pagliery, 2015b], it is necessary to develop techniques that minimize the aforementioned execution time $t_e(b)$. As a result, by constructing models that predict the shortest amount of dynamic analysis time required by each malware, it is possible to reduce the resources spent executing files whose harmful nature can be revealed in seconds. The time saved can then be reallocated to running binaries that demand longer periods of time and produce actionable intelligence for the analysis system [Vadrevu and Perdisci, 2016].

4.3 Approach

We want to build a machine learning-based model that determines malware analysis run time. To accomplish this, we propose a model that we refer to as TURACO (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs). TURACO can predict the approximate time needed to perform dynamic analysis on a file, in order to detect its malicious intent. This is a crucial factor, as dynamic analysis can provide valuable information, but it can be a bottleneck for analyzing malware. Traditional sandbox environments execute suspected files for long periods of time, often five minutes or more (see Table 4.1). In contrast, static analysis can often be performed in seconds. Furthermore, many malware do not need to be executed for a long time, because their malicious intent can be detected after observing their behavior for only a few seconds. TURACO leverages machine learning to predict the analysis time required to monitor each malware sample, instead of executing each file for a fixed amount of time. Effectively, our environment reduces the execution time dedicated to each malware binary (e.g., from 5 minutes to 10 seconds). These savings can be leveraged in multiple ways [Vadrevu and Perdisci, 2016]:

1. **Decrease the cost of malware analysis systems:** because fewer resources are needed to handle a predetermined daily volume of malicious files. Thusly, it is possible to design and provision a *bare-metal* architecture (i.e., a system running on real hardware) at a lower cost [Vadrevu and Perdisci, 2016].
2. **Significantly increase the capacity of malware analysis systems:** measured as the average number of malware samples that can be processed per day [Vadrevu and Perdisci, 2016].
3. **Spend less time analyzing samples from well-explored families:** and reallocate the time saved to running samples that require longer periods of time to display their malicious behavior, or are derived from truly new families [Vadrevu and Perdisci, 2016].

4.3.1 Features

The process of training TURACO involves using Radare2 to extract static analysis information from each unpacked malware that is being considered for dynamic analysis. Specifically, we extract byte-level representations of malicious code, as well as hashes histograms from each binary. These generated static features are then used to form part of our training dataset of static information. Analysis of static features of malware offers several unique benefits. First, it has the potential to cover all possible code paths of a malicious program, yielding more accurate representations of the entire functionalities of the malware [Nath and Mehtre, 2014]. Secondly, we intend to leverage the fact that approaches based on static features are much more scalable than their dynamic counterparts, as they do not require resource-intensive and time-consuming monitoring of malicious programs [Hu and Shin, 2013]. This is particularly important given the rapidly-increasing number of new malware samples. Finally, the goal of our system is not to exclude dynamic analysis approaches, but to complement the strengths of static analysis with dynamic analysis, to achieve higher accuracies in identifying the set of malicious files that require more expensive dynamic executions.

The remaining part of the dataset includes determining targets for each malware required for the supervised learning algorithms. Particularly, we leverage reports generated by Cuckoo that provide insights about the time required to confidently classify a file as malicious.

4.3.1.1 Byte Features

We provided an overview of byte-level features and bytes-entropy histograms in Chapter 2 (see Section 2.5.2.1). In this chapter, we use feature vectors that only consider 16 entropy bins (En) instead of 256 (as originally proposed in Figure 2.2), which reduces the length of the bytes-entropy histogram matrix by a factor of sixteen.

4.3.1.2 Hashes Histograms

We also introduced hashes histograms in Chapter 2 (see Section 2.5.2.2 and Figure 2.3). We extract the list of ASCII strings embedded in the code of a malware sample, as well as the import and metadata tables from the Portable Executable (PE) header. These strings are then hashed into fixed-size feature vectors of length 256.

4.3.1.3 Target Labels

The remaining part of our input features involves determining targets for each malware in our training dataset. We define a set of seven target classes: 10, 20, 30, 40, 50, 60, and 300 seconds. These labels indicate the minimum time each malware in the training set should be executed for, in order for its behavior to reach a minimum threat score of 5 within Cuckoo. Our choice for these targets is driven by the fact that they cover a wide number of execution times, ranging from very short executions (e.g., 10 and 20 seconds), to long executions (e.g., 60 seconds), to the traditional time used to run malware in an isolated environment (e.g., 300 seconds).

Furthermore, Table 4.2 shows the threat level scores for three different malware identified by their hash, and the seven different execution times corresponding to the seven target classes specified above. The score that determined the label that was assigned to each malware is highlighted in **bold**. The label corresponded to the minimum of our set target times that was needed for a threat level of 5 or higher to be reached. Each malware that was part of the training set was executed for each of the target times using Cuckoo, and a JSON report was generated at the end of each run containing the threat level score for the malware as highlighted in Figure 4.1.

Malware (m)	Scores for							Label (l)
	10s	20s	30s	40s	50s	60s	300s	
0d8e20309f0...802eba6f8b27532	**7.8**	7.8	8.0	8.4	8.4	8.4	8.4	10s
2d1a2e11a90...25dcd362277652	1.2	2.4	2.6	2.6	**5.4**	5.6	6.0	50s
2bae0de6cda...18d7f8c5536f142	0.4	1.0	1.0	1.6	1.8	2.4	**5.2**	300s

Table 4.2: Assignment of labels for malware dataset


```
{
    "info": {
        "added": 1528593608.859098,
        "started": 1528596558.067472,
        "duration": 114,
        "ended": 1528596673.001724,
        "owner": null,
        "score": 7.8,
        "id": 31,
        "category": "file",
        "git": {
            "head": "59d32361c1636b2b3802a1746f480a7768f6384f",
            "fetch_head": "59d32361c1636b2b3802a1746f480a7768f6384f"
        }
    }
}
```


Figure 4.1: This figure illustrates a malware threat's level score extracted Cuckoo.

4.3.1.4 Threat Level

A malware threat's level score can range from 1 to 10 based on the reports produced after multiple runs. It is also possible to determine that Cuckoo confidently classifies a file as malicious if the final score was equal or greater than 5.0. Quantifying the maliciousness of a malware was generated by Cuckoo after each analysis was completed. Furthermore, Cuckoo took an inventory of all the malicious behaviors that the malware exhibited. The developers of Cuckoo have a-priori identified a set of behaviors that could be deemed as malicious, such as installing itself for autorun during Windows startup, or querying for the computer name information. This set of behaviors were given a severity score, and the scores were combined into a single maliciousness scale from 1 to 10. Each malware is labeled with a target $t_e \in [10, 20, 30, 40, 50, 60, 300]$, signifying the minimum time needed to run the malware to receive a threat level score of 5.0.

4.3.1.5 Feature Engineering

We applied a series of transformations to make our predictive features more suitable for our classifiers. It has been demonstrated that some machine learning algorithms do not perform well when the input numerical attributes do not have a uniform scale [Brownlee, 2016]. An example of this phenomenon is found in the gradient descent algorithm used in logistic regression, support vector machine, and neural networks. If the input features are on different scales, certain weights may be updated faster than others, negatively affecting the convergence of the algorithm [Raschka, 2014]. As a result, we decided to *normalize* our input feature space by computing its mean μ and standard deviation σ and scaling each feature according to Equation 4.3 [Raschka, 2014].

$$z = \frac{x - \mu}{\sigma} \tag{4.3}$$

Moreover, we used a *logarithmic transformation (log)* to help stabilize the variances in our input data and handle possibly skewed distributions in our feature space [Sarkar, 2018]. Finally, the threat level of each malware and its normalized static analysis information are appended, to obtain the final training input that is fed to the supervised learning algorithms, as indicated in Table 4.3.

Hash (h)	Static Feature Vector (f)					Label (l)
	x_0	x_1	...	x_{n-1}	x_n	
fad4f2b0 ... 2ca8b839f	0.056	0.028	...	0.042	0.061	10s
8236581b ... e88db0e60	0.0233	0.1029		0.0861	0.0376	40s
⋮	⋮	⋮	⋱	⋮	⋮	⋮
6488913cf ... 148b5c14f	0.0933	0.0176	...	0.0561	0.0972	300s

Table 4.3: Training Dataset for TURACO model

4.3.2 Dataset

We obtained a data stream of 1.2 million binaries from Reversinglabs, which included malicious executables targeting financial services institutions. These malware

were derived from forty different families, designed to infect a variety of MS Windows versions including XP, 7, 8, and 10 for both 32-bit and 64-bit configurations. Additionally, we obtained a timestamp tag for each of these files indicating the date the malware was captured. These tags showed that our samples were gathered over a period of twelve years (2006 - 2018), with most of the binaries being collected in 2014 and 2016. Furthermore, our dataset was curated to guarantee the extraction of relevant static information from the malicious files. It is a well-known fact that most of the Windows malware are packed [Ugarte-Pedrero et al., 2015], and static analysis approaches fail at extracting meaningful features from malware. Our dataset provider removed all packing, obfuscation, and protection artifacts from the binary files to extract all internal objects with their metadata. As a result, the unpacked malware were available for further analysis using our disassemblers. Finally, for the experiments presented in this chapter, we selected malware families that included more than six thousand instances. As a result, our final dataset consisted of approximately 28336 files from five different families.

4.4 Learning Methodology

Figure 4.2 illustrates the architecture of our TURACO model. Each sample in our training set was executed for each of the target times using Cuckoo, and a JSON report was generated at the end of each run indicating the threat score for the malware. However, not all files reached the desired threshold to be recognized as malicious within the predefined time periods. This significantly reduced the size of our dataset (from 28336 to 20020 files), given that our system can only make predictions for executables that displayed malicious activities during their execution.

Figure 4.3 shows the final distribution of target labels extracted for the files in our dataset. In addition, it highlights the fact that it is indeed possible to run malware for a very short amount of time to reveal their malicious intent. In fact, Cuckoo was able to detect over 1979 malicious samples in just under 10 seconds. The number of files that required longer execution times, such as 300 seconds, only corresponded

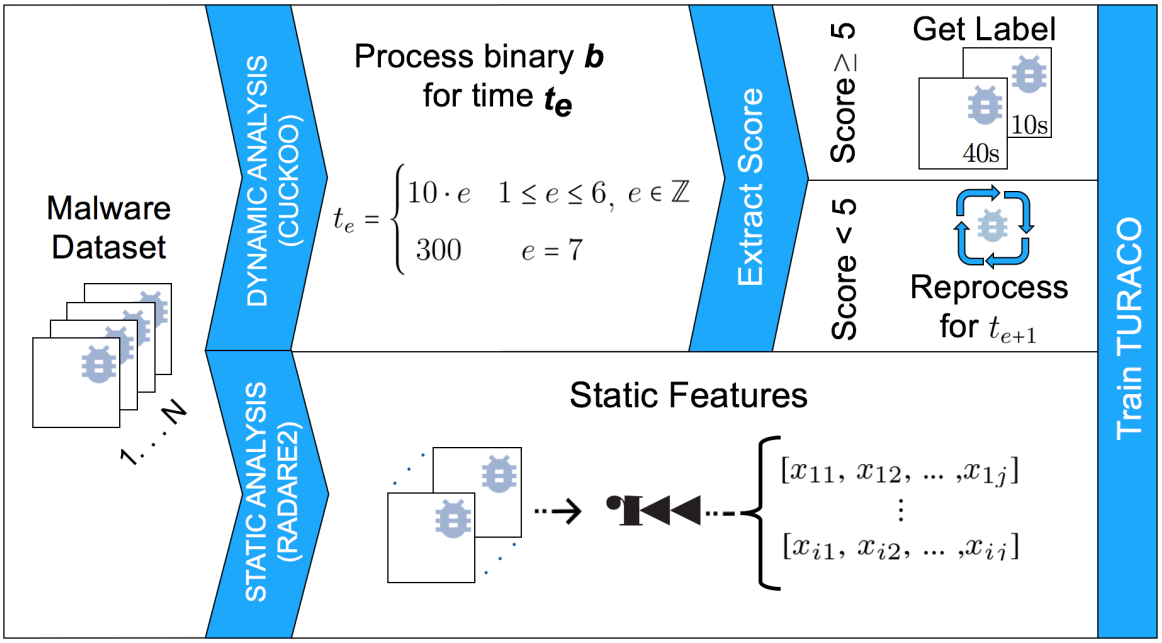

Figure 4.2: This figure describes the process of training our TURACO model. Dynamic analysis is performed in seven phases according to our predefined target labels of 10s, 20s, 30s, 40s, 50s, 60s, and 300s, until a threat score of 5.0 or greater is achieved. The minimum run time Cuckoo needed to reach a threat level score of 5 corresponds to the target label of our model. The label, together with the static features extracted from the malicious sample, becomes the training data for our learning algorithms.

to about 3977 samples in our dataset. This proves that it is not necessary to run an entire malware dataset for long periods of time (e.g., five minutes), because the malicious nature of many malware can be detected in only a few seconds. This is an important advancement in the field of malware detection, especially when we consider the narrow margin of time that security analysts have when confronted with malware such as ransomware, which can encrypt a user's data in a very short amount of time.

In addition, we constructed a cumulative histogram based on the aforementioned target labels. As indicated in Figure 4.4, our sandbox could classify over 6001 of the samples as malicious in 20 seconds or less, which corresponded to about 30% of our dataset. Moreover, a significant number of files were deemed malicious in 60 seconds or less (16040). Specifically, 80.13% of our dataset was determined as malware after only executing in the sandbox for one minute. This is indicative of the capabilities of

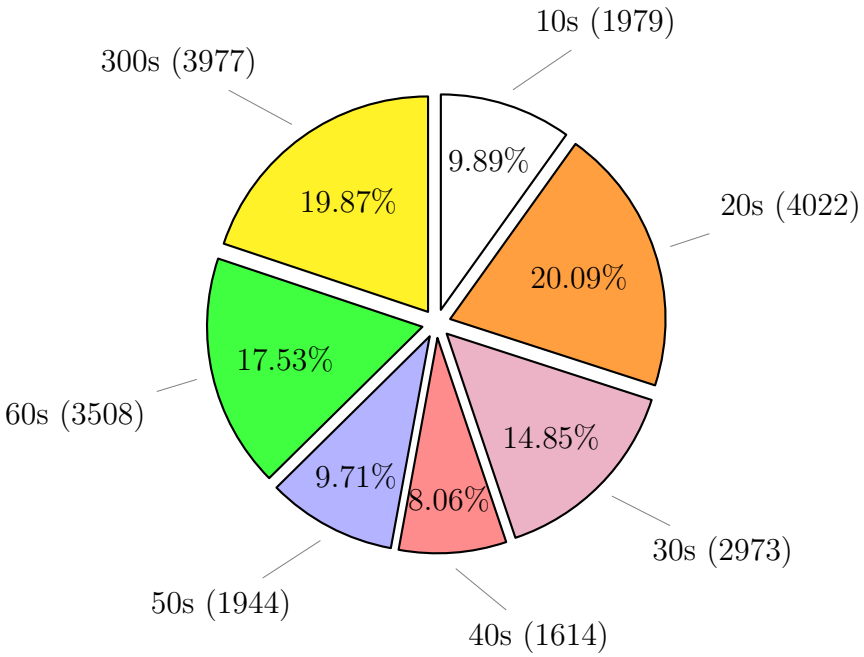

Figure 4.3: This figure shows the final distribution of target labels extracted for the malware in our dataset. It highlights the fact that the malicious nature of many malware can be detected in a short amount of time.

our system to analyze a file and detect that it is malware in a rapid manner.

4.5 Experimental Infrastructure

The deployment of TURACO was accomplished using *Sklearn*, an open source python library that implements a wide range of machine learning algorithms for data analysis [Pedregosa et al., 2011]. Four particular classifiers were selected to test our approach, namely: decision tree (DT), support vector machine (SVM), artificial neural network (ANN), and random forest (RF). In addition, we used *nested cross validation* to perform the training, validation and testing of our algorithms [Albon, 2017]. This technique divides the data set into five equally-sized subsets, and runs five experiments in which three of the subsets are used for training, one for validation, and the final subset for testing. Moreover, we perform *stratification* in our experiments. Stratification enables us to randomly select malware from the dataset so that the test set

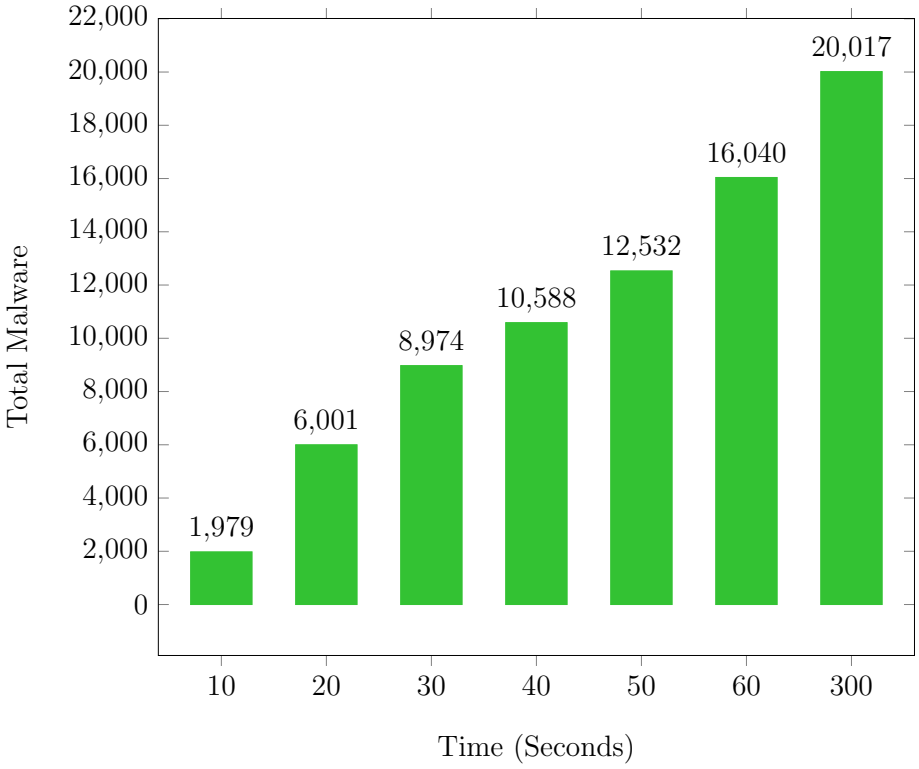

Figure 4.4: This figure shows a cumulative histogram of the run time labels extracted from Cuckoo. Our sandbox could classify over 6001 binaries, or 30% of our malware corpus, as malicious in 20 seconds or less. This is indicative of the capabilities of our system to analyze a file and detect that it is malware in a rapid manner.

contains the same distribution of instances among the target classes as the overall data set [Dwinnell, 2007].

4.6 Experimental Results

In this section, we discuss the major results from the implementation of our TURACO model using Sklearn. First, we discuss the accuracy scores obtained by each of our classification models. Then, we evaluate the effectiveness and completeness of TURACO by analyzing its recall and precision results, as well as its confusion matrix.

4.6.1 Accuracy Results

As seen in Figure 4.5, the random forest classifier yields the highest accuracy (93.86%), followed by artificial neural network (91.51%), decision tree (90.41%), and support vector machine (89.94%). These results are clearly indicative of the ability of machine learning algorithms to estimate the time that malware should be executed in a sandbox. Additionally, we confirm that we can indeed use fast static features extracted from suspected samples to estimate the malware analysis run time, based on the seven predefined target classes of 10, 20, 30, 40, 50, 60, and 300 seconds.

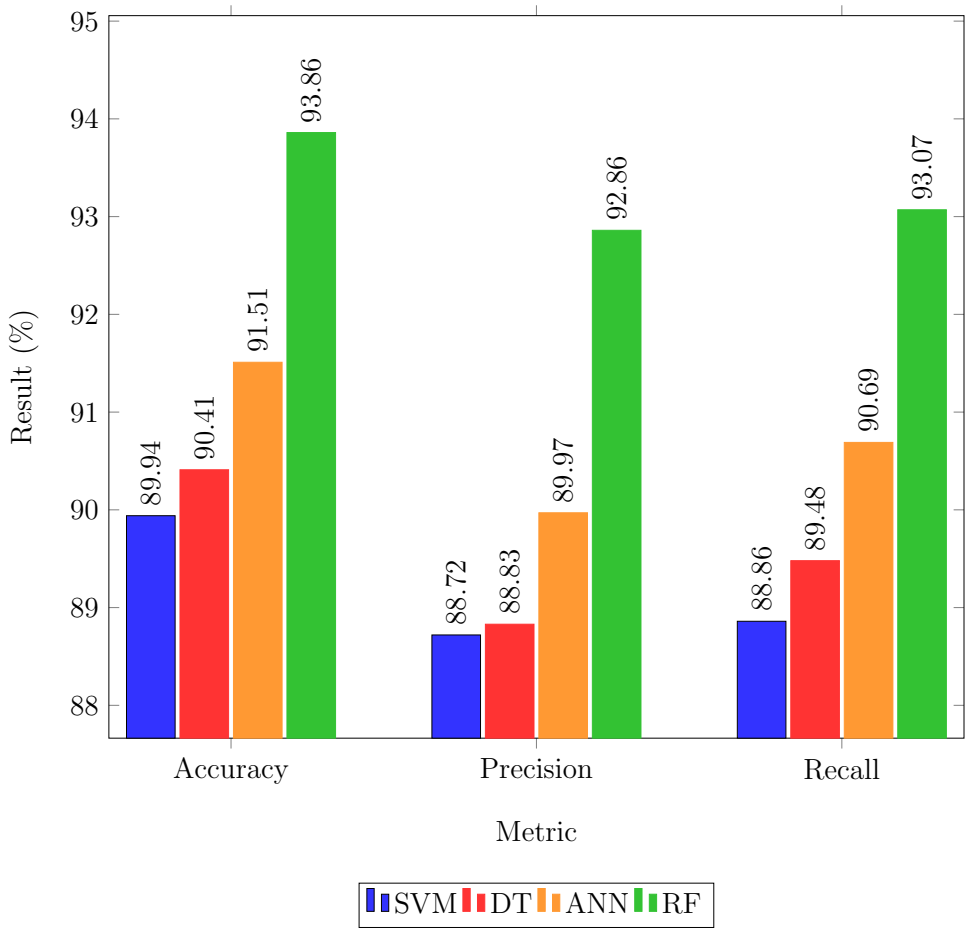

Figure 4.5: This figure depicts the accuracy, precision, and recall results for our machine learning classifiers. The implementation of our TURACO model using the random forest classifier yields the highest accuracy (93.86%). In addition, the recall and precision results are fairly consistent across our four machine learning models, which is indicative of the completeness of our classifiers.

If our model predicts a conservative stopping point (e.g., 300 seconds), it is simply defaulting to a fixed amount of time which is what traditional sandbox environments do. However, TURACO often predicts to run the malware for shorter amounts of time and is almost 94% accurate at correctly predicting the time needed to execute the malware.

4.6.2 Recall and Precision Results

In addition to the accuracy scores, we examined the precision and recall produced by our machine learning classifiers. Given that the approach proposed in this chapter considers a wide range of run times, ranging from short executions (e.g., 10 seconds), to long analysis run times (e.g., 300 seconds), our models need to be able to identify these *critical* classes to be considered effective. Figure 4.5 shows that the recall and precision results are fairly consistent across our four machine learning models. To further expand on these results, we plot the confusion matrix for our best-performing model (i.e., random forest) and compute the recall and precision for each class label.

Table 4.4 depicts the confusion matrix for our TURACO model based on the random forest algorithm. The large values in the diagonal of the matrix are indicative of the high predictive power of TURACO to determine malware analysis run times. Moreover, most of the misclassifications of our model corresponded to pairs of non-critical adjacent times such as 30 seconds and 40 seconds or 50 seconds and 60 seconds.

		Predicted						
		10s	20s	30s	40s	50s	60s	300s
Actual	10s	**370**	14	9	0	2	0	1
	20s	19	**759**	8	5	10	2	1
	30s	8	17	**550**	13	0	5	2
	40s	0	1	11	**290**	13	4	4
	50s	5	3	2	9	**345**	24	1
	60s	0	1	6	4	21	**668**	2
	300s	2	2	0	3	5	7	**776**

Table 4.4: Confusion Matrix for TURACO Model

Additionally, the recall results in Table 4.5 highlight the completeness of our model, as it can return most of our critical classes (i.e.,10, 20, and 300 seconds) and yield a low number of false negatives for the samples in our dataset. This is a crucial factor of the analysis system, as long periods of time, such as five minutes, are not spent analyzing samples that only require a few seconds to reveal themselves (e.g., 10 or 20 seconds). Similarly, the label of 300 seconds is correctly assigned to a significant number of files that need longer times and, therefore, more computational resources from the analysis system to reveal their malicious intent.

	Run Time Label						
	10s	20s	30s	40s	50s	60s	300s
Recall (%)	93.43	94.40	92.44	89.78	88.69	95.16	97.61
Precision (%)	91.58	95.23	93.86	89.51	87.12	94.08	98.60

Table 4.5: Individual recall and precision results for TURACO model.

4.7 Discussion

In this chapter, we aimed to answer the following two questions: Can we run a file for a *short* amount of time and still detect that it is malware? Can we use features extracted from *fast* static analysis to predict the time that malware should be run in a sandbox environment? We address the first question by analyzing the results obtained by running our initial malware dataset using Cuckoo.

Our findings show that we can indeed execute malware in a sandbox for a very short amount of time and still collect enough information to deem them as malicious. In fact, our results demonstrate that nearly 80% of the samples in our dataset can be identified in less than 60 seconds and only 20% need to be executed for long periods of time. This is a crucial finding as it enables us to reduce the time spent on dynamic analysis, and reallocate the time saved to analyzing samples that actually improve the intelligence of our malware analysis system.

To answer the second question, we examine the results obtained from the implementation of TURACO. Our findings indicate that a random forest algorithm trained

on fast static characterization of malware can lead to accurate run time predictions. TURACO is able to distinguish between malware that need long execution times, from malicious executables whose harmful nature can be detected in a time as short as 10 seconds. This allows us to increase the throughput of our analysis system as we decrease the time that is required for expensive and slow dynamic analysis of malware. This claim is further supported in the next section.

4.7.1 Efficiency of TURACO

In this section, we analyze the time that was saved by using our TURACO model compared to another approach used to perform dynamic analysis. For the purpose of this study, we used the test dataset for which our model predicted the malware analysis time. Additionally, we defined the following two methods to perform dynamic analysis on the test dataset.

1. **Using maximum amount of time:** the entire malware test dataset was analyzed for the predefined time of 300 seconds.
2. **Using predictions made by TURACO:** our prediction model was used to determine the times that each sample of the test dataset should be executed in our analysis system. If the threshold of 5.0 was not achieved after running the sample according to the predicted labels, the malware was resubmitted for analysis for the next predefined time until the desired score was obtained.

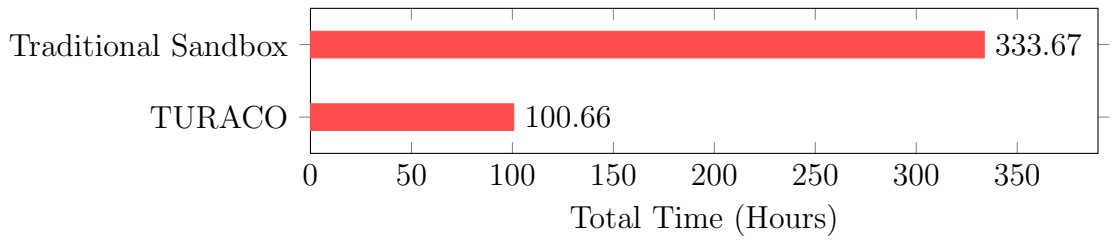

Figure 4.6: This figure depicts the time saved by using our TURACO model compared to a traditional dynamic analysis approach where malware is executed for 300 seconds. TURACO requires the least amount of time (100.66 hours) and reduces the average execution time that is required to uncover the malicious intent of our test dataset by 232.01 hours or 69.53%.

As indicated in Figure 4.6, TURACO requires the least amount of time (100.66 hours), compared to the second approach for dynamic analysis of malware (333.67 hours). Specifically, our prediction model reduces the average execution time that is required to uncover the malicious intent of our test dataset by 232.01 hours or 69.53%, when compared to a method where all the samples are submitted for a total of 300 seconds or 5 minutes, which corresponds to the approach that traditional sandbox environments use to examine malware behavior [Vadrevu and Perdisci, 2016].

4.8 Related Work

Existing approaches using machine learning and dynamic data to detect and classify malware use the entire execution cycle of the suspected samples, regardless of the information extracted from the files in question. This creates an impractical delay (in comparison with static analysis) if dynamic analysis is used as part of an endpoint security system [Rhode et al., 2018]. As a result, a few works have been proposed to improve the efficiency of dynamic malware analysis. In the following sections, we examine and compare the results found in this chapter to two particular approaches proposed in the literature.

Bayer et al.[Bayer et al., 2010] attempted to improve the efficiency of dynamic malware analysis by developing a technique that reduced the overall malware analysis time. Their solution was based on the insight that a large number of new malware is due to the mutations of only a few malicious programs. To avoid performing a full dynamic analysis of the same polymorphic malware, the authors analyzed binaries by executing them for a predefined time of 45 seconds. The next step involved checking whether or not the behavior seen during this time frame is similar to an already analyzed file. If that was the case, the authors interrupted the analysis of the file and, instead, leveraged the information from the existing behavioral profile to characterize the suspected sample. This approach was tested on a set of 10922 files and the authors managed to avoid a full dynamic analysis of over 2747 files (nearly 26% of their dataset). Nevertheless, this technique is less effective that the one we propose in this

chapter. First, bad authors can craft two binaries to appear to have identical behavioral profiles at the 45-second check point, and display completely different malicious activities afterwards. In addition, their method can also be evaded if the executables need additional time to reveal their true intent [Bayer et al., 2010].

Similarly, Vadrevu and Perdisci [Vadrevu and Perdisci, 2016] proposed a probabilistic multi-hypothesis testing framework to speed up the dynamic analysis of malware. Their approach attempted to determine whether a malware sample that was submitted for analysis in a sandbox environment had likely been examined before, and its execution could be preemptively stopped [Vadrevu and Perdisci, 2016].

First, the authors clustered malicious samples into groups or families that displayed similar network activities (e.g., equivalent DNS queries or HTTP requests) [Vadrevu and Perdisci, 2016]. Then, a behavioral profile was constructed summarizing the network events generated by all the samples in the family. The next step involved extracting dynamic information from a set of malware files. The authors computed the probability that the sample belonged to an already seen family, given all the events generated by the file until a specific execution time. If the probability was greater than a predefined threshold, the execution was stopped and the sample was assigned to the family with equivalent network behavior. Otherwise, the files were allowed to continue their execution and for every new network event, the probability of being part of an existing family was recomputed until the maximum amount of time (360 seconds) was reached [Vadrevu and Perdisci, 2016].

This technique was tested on two datasets of 15,431 and 62,063 samples and their results showed that the average stopping times for both sets were 69.9 seconds and 50.4 seconds, respectively [Vadrevu and Perdisci, 2016]. This method, however, fails if a malware author prepends network events of a known malware family to a new malicious sample. Their platform then may decide that a new malware belongs to an already analyzed family, and prematurely stop its dynamic execution. In addition, the approach proposed by Vadrevu and Perdisci relies exclusively on network information [Vadrevu and Perdisci, 2016]. Therefore, it cannot successfully determine the time required to

analyze a malicious sample, if the file does not exhibit any network activities.

TURACO, on the other hand, does not necessarily depend on any specific type of dynamic information, and considers all the possible malicious activities that a malware can exhibit during its execution in a sandbox environment. Furthermore, it leverages fast static analysis information to predict the time at which to stop the execution of the suspected files. The approaches proposed by Vadrevu and Perdisci, as well as Bayer et al, attempted to reduce the malware execution time by employing dynamic features only, which is inherently slower than our approach, as it still requires the execution of the suspicious samples in an isolated environment. Our model bases its prediction on static features of unpacked malware, which are extracted in seconds, without the need to perform expensive dynamic analysis. Finally, TURACO is able to detect a significant number of malware in just under ten seconds, outperforming the average stopping times of the previously discussed approaches.

4.9 Conclusions

In this chapter, we investigated the application of machine learning techniques to the problem of predicting malware analysis run time. We introduced a model called TURACO (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs) that significantly extends the capabilities of Cuckoo, an open-source sandbox. TURACO leverages features extracted from fast static analysis of malware, as well as reports generated by Cuckoo to predict malware analysis run time.

Our results demonstrated that TURACO could successfully identify the amount of time that malware should be executed for in a sandbox in order to reveal itself. This execution did not require a significant amount of time, as seen from the fraction of our dataset that was deemed as malware in under ten seconds. Additionally, we showed that it was possible to use static features to construct highly accurate machine learning models that estimate malware analysis run time.

Finally, we evaluated the efficiency of our proposed model. TURACO drastically reduced the average execution time required to uncover the malicious intent of our test

malware dataset. In fact, a reduction in execution time of over 69.53% (or 232.01 hours) was achieved by our model, when compared to a method where all the samples were submitted for a total of 300 seconds, which corresponds to the approach that traditional sandboxes use to examine malware behavior.

Chapter 5

SEEMA: SELECTING THE MOST EFFICIENT AND EFFECTIVE MALWARE ATTRIBUTES

5.1 Introduction

Dynamic analysis is an effective characterization of malware that allows security analysts to extract meaningful runtime information from the execution of binaries in a sandbox environment. As shown in the Chapter 2, Section 2.7, features extracted from dynamic analysis reports, such as system call sequences, can lead to models that yield high accuracy rates for the problem of malware classification. Moreover, dynamic analysis is less susceptible to obfuscation techniques designed to thwart static analysis and, as a result, can successfully examine encrypted and packed malware [Hu et al., 2013].

Yet, on other malware, applying dynamic analysis has very little impact and may lead to an inaccurate prediction in some cases. This can be explained from the fact that dynamic analysis has limited coverage of the behavior of the malware as it only captures the API calls, registry changes, and network activities that correspond to the execution path taken during a particular run of the sample [Hu and Shin, 2013]. In fact, various execution paths may be taken depending on the program's internal code and configuration of the analysis environment. For example, some malware stall the execution of any malicious activity if they do not detect any user interaction with the analysis environment. Other malware perform certain malicious operations only under specific conditions (e.g., a particular date or time is reached) [Hu et al., 2013].

In contrast, static analysis can characterize all possible code paths of a malicious sample, including those that are not normally executed [Nath and Mehtre, 2014].

Therefore, static analysis can lead, in some instances, to better characterizations of malware and more accurate and robust classification models.

Even so, if dynamic analysis were an inexpensive characterization of malware, we would apply it to all binaries, regardless of whether or not it leads to an accurate prediction. However, performing dynamic analysis is computationally expensive because it involves executing the suspected samples for a significant amount of time (e.g., five minutes or more). Because it is costly and does not always improve the classification of malware, we want to apply dynamic analysis selectively.

In this chapter, we first introduce a set of three models of increasing cost and predictive power for malware classification. These models are constructed based on information extracted from both static and dynamic analysis of malware. We then develop a surrogate model called *SEEMA* (**S**electing the most **E**fficient and **E**ffective **M**alware **A**ttributes) that correctly predicts when less expensive characterizations, such as static analysis, will suffice to accurately classify malware. This allows us to *filter* the binaries that will *not* benefit from time-consuming executions in an emulated environment. Given that in practice a large portion of malware does not require expensive-to-compute dynamic features, and static analysis is orders of magnitude faster than dynamic analysis, we significantly decrease the time spent extracting and engineering features for malware classification.

5.2 Problem Statement

Assigning malicious samples to their corresponding families is one of the most challenging problems in cyber security. *Malware family classification* describes the process of determining whether a binary is part of a previously seen family or if it corresponds to a new unseen sample that requires further examination [AlAhmadi and Martinovic, 2018].

A malware family provides security analysts with crucial information about the threat level that a file poses to the system as well as the remediation process that needs to be followed to mitigate a potential threat [Kaspersky, 2018]. For example,

the incident response for ransomware differs from a botnet's mitigation process, as the former represents a larger threat to a system and must be prioritized accordingly [AlAhmadi and Martinovic, 2018]. Additionally, malicious programs derived from the same code base bear significant similarities, and this consistent behavior can thus be leveraged to identify malware families [Hu et al., 2013].

In recent years, several approaches have been proposed to solve the problem of predicting malware families. As described in Chapter 2, techniques based on static and dynamic features extracted from malicious code can lead to highly accurate classification models. However, the selection of relevant features to feed as input to a classifier remains a complex task. In addition, training predictive models based on dynamic-based features is not efficient compared to static analysis. To support this claim, we monitored the time required to extract two different static characterizations of malware for datasets of varying sizes, namely: 1) byte-level analysis, and 2) graph-based features. The results were compared against the standard time required to perform dynamic analysis (i.e., three hundred seconds), and our findings are summarized in Figure 5.1.

The time required to generate byte features generally depends on the size of the file. Nevertheless, they are extremely efficient and can be constructed using optimized C code. As seen in Figure 5.1, byte features can be generated in less than a second, even for large malware datasets. Graph-based features, on the other hand, take longer to generate because our tool needs to extract assembly code from the file and transform it into graphs at three levels of granularity (i.e., functions, blocks, and operations) from which we construct feature vectors. This expensive static characterization of malware, however, still remains more efficient than performing dynamic analysis as it only demands 77 seconds for large malware datasets, in contrast to the 300 seconds usually required to execute a file in a sandbox.

Furthermore, Figure 5.1 shows that dynamic analysis is nearly 300 times slower than byte-level analysis of malware. This highlights the importance of leveraging cheap-to-compute features for malware family classification and identifying when dynamic

analysis leads to more accurate prediction models.

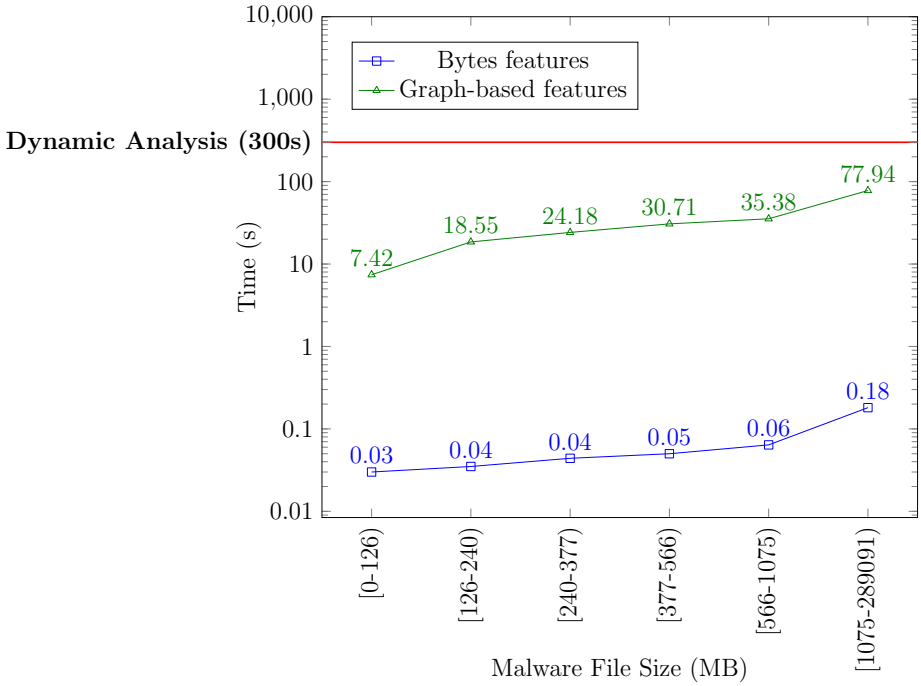

Figure 5.1: This figure shows the cost of producing three characterizations of malware. Analysis of malicious software at the byte-level usually takes less than a second to generate. Graph-based features, on the other hand, take longer to produce because our tool needs to extract assembly code from the file and transform it into graph-like data structures, from which we construct feature vectors. Finally, dynamic analysis often demands the execution of malware for over 5 minutes or more, making it less efficient for the characterization of large malware datasets.

5.3 Approach

We want to construct a machine-learning based model that predicts when using dynamic-based features benefits the classification of malware over simpler characterizations, such as static analysis. To accomplish this, we propose *SEEMA*, a surrogate model that **s**elects the most ***e***fficient and ***e***ffective ***a***ttributes of ***m***alware. SEEMA can choose from three malware classification models constructed using features of increasing cost and predictive power, namely: 1) byte-level analysis, 2) graph-based features, and 3) dynamic-based features. As seen in Figure 5.2, SEEMA acts as a filter that

reduces the feature engineering process and saves execution time. Our model only extracts the most costly features for a small fraction of a malware corpus when it deems that the additional benefit of using dynamic analysis is worth the effort.

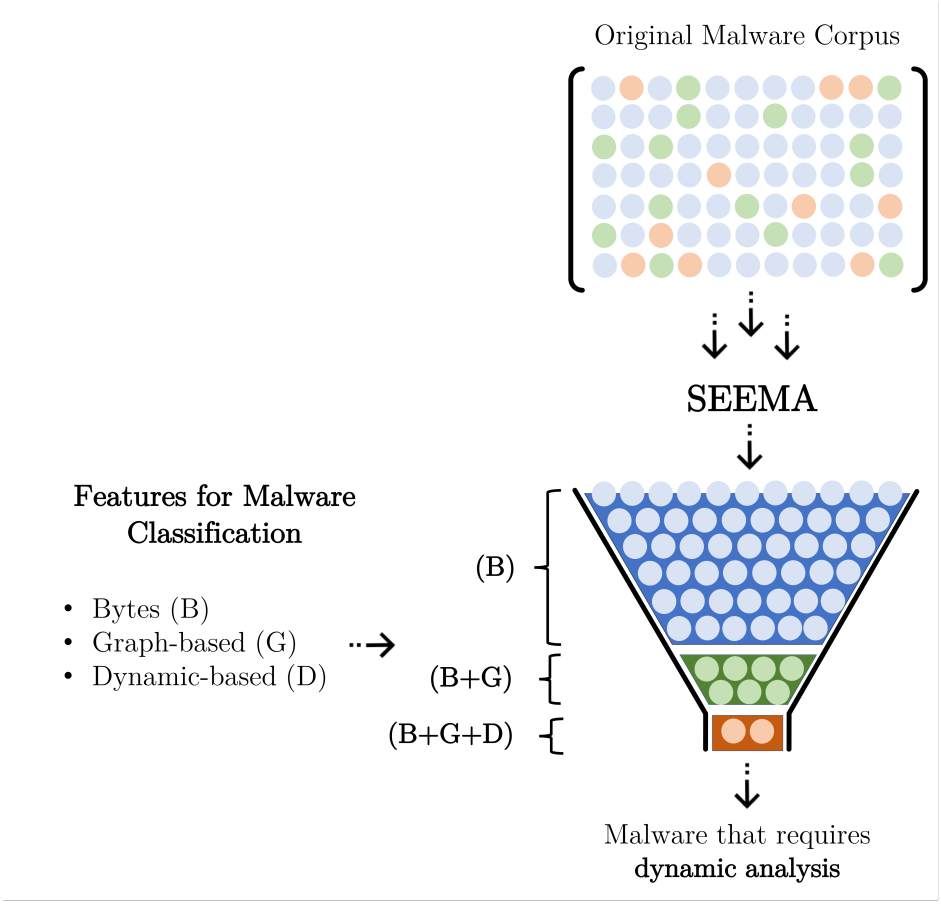

Figure 5.2: Our SEEMA model acts as a filter that decreases the time dedicated to feature engineering. SEEMA only extracts the most expensive features for a small subset of a malware corpus when it deems that the additional benefit of using dynamic analysis is worth the effort.

5.3.1 Features

First, we explore the selection of relevant features from both static and dynamic analysis. Specifically, there are three main categories of features that we investigate:

1. Traditional computationally-inexpensive byte-level representations of malicious code.

2. Computationally-expensive statically analyzed disassembled instructions.
3. Features extracted from the dynamic execution of malware in a sandbox environment.

5.3.1.1 Byte Features

We presented a description of byte-level features and bytes-entropy histograms in Chapter 2 (see Section 2.5.2.1). For the feature vectors used in this chapter, we only consider 16 entropy bins *En* instead of 256 (see Figure 2.2), which decreases the length of the bytes-entropy histogram matrix by a factor of sixteen.

5.3.1.2 Graph-based Features

We described graph-based features in Chapter 2 (see Section 2.5.2.3). We use Radare2 to translate a malware executable into assembly code and construct graph-data structures at three levels of granularity: instruction-flow graph (instructions), control-flow graph (blocks), and call graph (functions). We then use graph spectral analysis to build fixed-size representations of the graphs. Particularly, we construct adjacency lists, which aggregate information about the nodes and edges of the graph, and an instruction histogram for each node in the graph. For our experiments, we choose to leverage the block level information of the malware since it is more manageable in size than the instruction level and usually more expressive than the function level.

5.3.1.3 Dynamic Features

As described in Chapter 2, dynamic analysis of malware is performed using Cuckoo. We monitor the activities of malicious executables in a sandbox environment for a total of five minutes and extract their behaviors in JSON files. These reports contain a section called *apistats*, which summarizes the number of times a particular set of API calls were executed during the analysis of the file (see Figure 5.3). Finally, we leverage this information to produce a histogram of API Calls that is used to train our malware family classifiers.

```
{
    "behavior": {
        "apistats": {
            "2360": {
                "LdrUnloadDll": 1,
                "GetSystemInfo": 1,
                "GetSystemWindowsDirectoryW": 1,
                "NtClose": 8,
                "GetFileAttributesW": 3,
                "NtFreeVirtualMemory": 1,
                "CreateActCtxW": 4,
                "SetErrorMode": 3,
                "NtAllocateVirtualMemory": 7,
                "NtWriteFile": 1,
                "LdrGetDllHandle": 36,
                "NtOpenFile": 1,
                "GetSystemDirectoryW": 1,
                "SetUnhandledExceptionFilter": 1,
                "NtCreateFile": 1,
                "GetSystemTimeAsFileTime": 4,
                "FindFirstFileExW": 3,
                "NtCreateMutant": 1,
                "NtProtectVirtualMemory": 6,
                "LoadStringW": 2,
                "NtOpenMutant": 1,
                "LdrGetProcedureAddress": 50,
                "NtOpenKeyEx": 1,
                "LdrLoadDll": 2,
                "NtQueryInformationFile": 1,
                "CreateProcessInternalW": 1
            }
        }
    }
}
```


Figure 5.3: This figure depicts a snippet from a report generated by Cuckoo, which summarizes the number of times a set of API calls were executed during the analysis of the file.

5.3.1.4 Feature Engineering

As described in Chapter 4, Subsection 4.3.1.5, some machine learning algorithms do not perform well when the input numerical attributes do not have a uniform scale [Brownlee, 2016]. Therefore, we decided to *normalize* our input feature space and use a *logarithmic transformation (log)* to help stabilize the variances in our input data and handle possibly skewed distributions in our feature space [Sarkar, 2018].

5.3.2 Dataset

As mentioned in Chapter 4, Subsection 4.3.2, we obtained a data stream of 1.2 million binaries from Reversinglabs; this data stream included malicious executables

targeting financial services institutions. For the experiments presented in this chapter, we selected the malware families that included more than two thousand instances. As a result, our final dataset consisted of 30315 files from nine different families. Table 5.1 provides a breakdown of our malware dataset, including the count and type for each family.

Name	Count	Type
Andromeda	2220	virus
Shifu	2255	spyware
Inject	3405	trojan
Cutwail	3435	downloader
Zbot	3470	downloader
Ramnit	3565	trojan
Injector	3850	trojan
Banker	4045	spyware
Banload	4070	downloader

Table 5.1: Composition of dataset for malware family classifiers

5.4 Learning Methodology

5.4.1 Malware Family Classifiers

We generate our three malware classification models using a set of features of increasing cost and predictive power. Particularly, our first classifier (B) only includes features extracted from byte-level representations of malware. The second classifier that we propose (B-G), leverages the previous byte-level information as well as graph-based features to predict malware families. Finally, our most expensive classifier (B-G-D) takes advantage of a hybrid feature set that includes both static information (i.e., byte and graph-based features) and dynamic information.

To build these malware classification models, we use a method known as *nested cross-validation* [Albon, 2017]. This technique uses a series of train/validation/test set splits, where the validation set is used to tune the parameters of the algorithm and select the best configuration, and the testing fold provides an unbiased evaluation of the model. Consequently, we first split our dataset into five stratified folds. *Stratification*

allows us to generate training, validation, and testing folds that contain a distribution of class values consistent with our dataset [Dwinnell, 2007]. We use three folds for training, one fold for validation, and the remaining fold for testing. This breakdown enables us to evaluate the ability of our models to generalize what was learned on a training set to a testing set. In addition, it guarantees that all the files in our dataset are used for testing exactly once. This is a vital step because we need to determine whether or not our three classifiers yield an accurate prediction for the classification of malware. This information is leveraged to create the input features for our SEEMA model.

Given a malware sample m, we aggregate information about the predictions made by each of our classification models B, B-G, B-G-D. A value of "1" indicates that a classifier c generated a correct prediction for a malware file m, and a value of "0" designates a misclassification. This process is summarized in Table 5.2.

Malware (m)	Classifier (c)			Label (l)
	B	B-G	B-G-D	
abb16d8c . . . d2ef8f37fe	0	**1**	1	B-G
8236581b . . . e88db0e60	**1**	1	1	B
⋮	⋮	⋮	⋮	⋮
6488913cf . . . 148b5c14f	0	0	**1**	B-G-D

Table 5.2: Construction of input dataset for SEEMA model.

Furthermore, if more than one classifier produces an accurate classification for a malware m, we select the most inexpensive model as the label for the sample. For instance, the second row of Table 5.2 shows a malware binary that was correctly classified by all of our models. A target label of "B" is then assigned to this sample as it corresponds to the simplest characterization required to produce a correct classification.

5.4.2 SEEMA Model

As mentioned earlier, the results from our three malware classifiers are used to generate the input features for our SEEMA model. In addition, this feature space includes byte-level information extracted from the files in our dataset. We chose bytes

because they correspond to the fastest static characterization and can be generated in an average of 0.06 seconds. This information is combined with the target labels describing the most inexpensive model required to classify the sample to form the final training instances, as indicated in Table 5.3. Finally, SEEMA is deployed using an approach similar to the one previously described to build our malware classifiers (i.e., five stratified folds, where three are used for training, one is used as a validation set, and the last one gauges the effectiveness of the model).

Malware (m)	Byte Features					Label (l)
	x_0	x_1	...	x_{n-1}	x_n	
abb16d8c . . . d2ef8f37fe	0.0421	0.0609	. . .	0.0715	0.0238	B-G-D
8236581b . . . e88db0e60	0.0233	0.1029	. . .	0.0861	0.0376	B
⋮	⋮	⋮	⋱	⋮	⋮	⋮
6488913cf . . . 148b5c14f	0.0933	0.0176	. . .	0.0561	0.0972	B-G

Table 5.3: Input dataset for SEEMA model

5.5 Experimental Infrastructure

We implemented our classifiers using Sklearn's logistic regression module [Pedregosa et al., 2011]. Our choice of logistic regression was driven by the highly interpretable nature of its output, as it can highlight the probability that a malicious sample belongs to each of the target classes considered in our experiments. Moreover, given that our classifiers must make predictions for nine different malware families, we implemented the logistic regression classifier in its multiclass form, which makes use of the *one-vs-rest (OvR)* scheme. This particular implementation involves creating a binary classifier $h^{(i)}(x)$ for every class i in our dataset. The OvR strategy then transforms the values of a class i into positive samples (i.e., they are assigned a value of "1"), and the rest of the classes are turned into negative samples (i.e., they are given a value of "0"). Given an instance x and parameter θ, each binary classifier estimates the probability that the target label y of the instance is equal to class i, i.e., $P(y = i|x; \theta)$. This results in a vector $h_\theta(x) = [h^1_\theta(x), h^2_\theta(x), ..., h^n_\theta(x)]$, where n corresponds to the

number of classes. Finally, to make a prediction for a new sample x, the algorithm selects the class i that maximizes $h_\theta^i(x)$.

5.6 Experimental Results

In this section, we evaluate the effectiveness of static analysis as compared to dynamic analysis for the problem of malware classification by comparing the performance of our three classifiers in terms of accuracy, precision, and recall.

5.6.1 Malware Classifier Results

5.6.1.1 Accuracy Results

The accuracy rates obtained by each of our three malware classifiers are presented in Figure 5.4. In addition, we show the accuracy results obtained for each class in Figure 5.5, in order to evaluate the individual performance of our models across malware families.

The malware classification model B-G-D, which was constructed using hybrid features (i.e., dynamic and static analysis information), produced the highest accuracy out of all three malware family classifiers (93.02%). However, the B model, which was trained on simple byte-level information, was able to generate model results for the classification of malware. In fact, it achieved an average accuracy of over 80.37%, which was partially affected by certain malware families. As seen in Figure 5.5, our B model was able to accurately classify malware as cutwail, zbot, ramnit, shifu, and andromeda, but did not perform well in terms of the injector, inject, banker, and banload families.

This phenomenon can be explained from existing connections among these malware families. Banload corresponds to a family of trojans that downloads other malware onto a user's machine, typically members of the *banker* family [Microsoft, 2018]. Similarly, both inject and injector are part of a family of trojans designed to insert malicious code into processes running on a computer in order to modify registry information (as in the case for the *inject* class) or download other malware and monitor

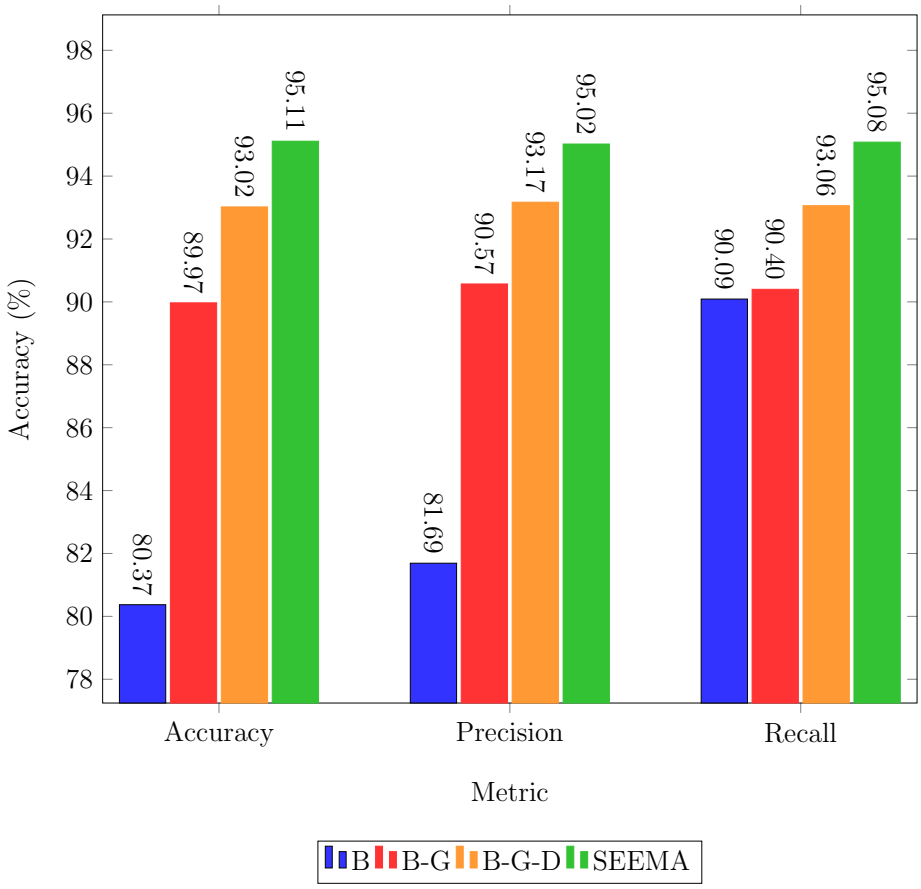

Figure 5.4: This figure shows the accuracy, precision, and recall results for our malware classifiers and SEEMA model.

a user's actions (as done by the *injector* family). These similarities might explain the inability of our model to make accurate predictions for these four classes.

On the other hand, the accuracy rate of 80.37% achieved by our B model highlights how a significant portion of our malware dataset can be correctly assigned into their corresponding families using only cheap-to-compute features. This result is further supported by Figure 5.6, where nearly 82.16% of the samples in our dataset were correctly classified using simple byte features. This value is slightly larger than the overall accuracy of the B model because it also includes samples for which no model yielded a correct classification. Moreover, adding more computationally expensive static analysis (B-G) and dynamic analysis (B-G-D) only improved the accuracy scores for 9.67%

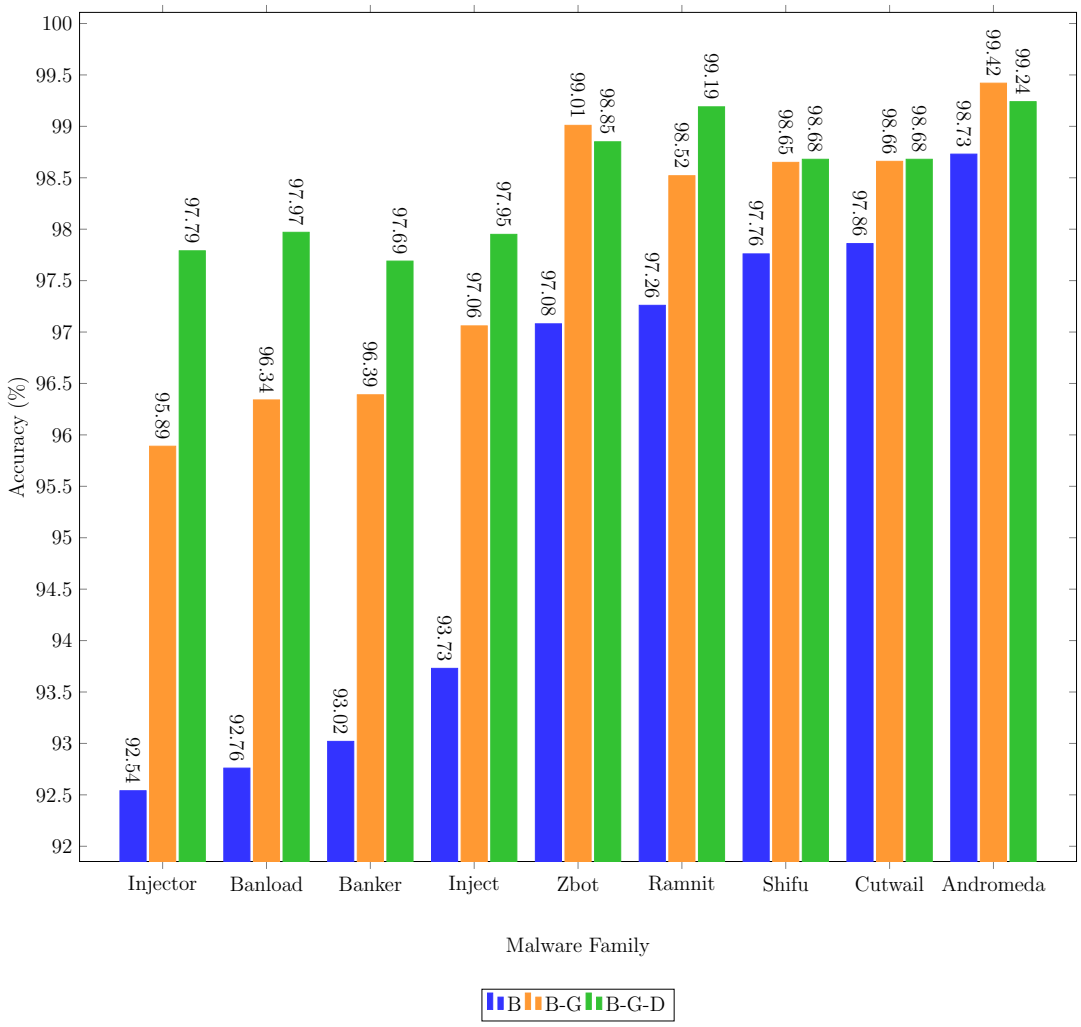

Figure 5.5: Individual accuracy results for malware classification models.

and 8.17% of the binaries in our dataset, respectively.

5.6.1.2 Precision and Recall Results

We also computed precision and recall for each malware family to further evaluate the performance of our classifiers. As indicated in Figure 5.7 and Figure 5.8, the precision and recall results generally improve as we use more computationally expensive features to predict malware families.

However, in some cases, our most expensive model (B-G-D) does not necessarily yield the best results. For instance, the precision results indicate that given all the

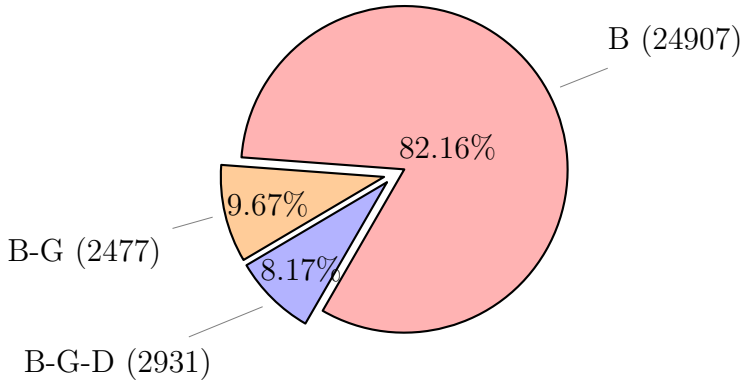

Figure 5.6: This figure shows the distribution of features required to classify the samples in our dataset. 82.16% of the files only need simple byte features. Furthermore, adding more computationally expensive static analysis (B-G) and dynamic analysis (B-G-D) only improves the accuracy scores for 9.67% and 8.17% of the binaries, respectively.

samples that were labeled as being part of the shifu, zbot, and andromeda families, our B-G-D model was only correct 90.27%, 97.71%, and 96.63% of the time, respectively. In contrast, our B-H-G model managed to achieve better precision scores of over 91.46%, 97.74%, and 97.01% for the same families, respectively.

The most important contribution of our hybrid model corresponds to the accurate identification of the inject, injector, banload, and banker families. The results for these families gradually improved as we used more expensive features to train our machine learning models. However, it was our B-G-D model that showed the greatest improvement in precision and recall, largely outperforming the results attained by our static-based classifiers.

5.6.2 SEEMA Model Results

As explained in Section 5.4.2, we use the output from our classifiers to create the input dataset for our SEEMA model. Nevertheless, as described in the preceding paragraphs, a significant segment of this malware corpus (82.16%) was labeled as needing only the simplest static features to be correctly classified. Using all the available observations from this imbalanced dataset would significantly affect the quality and

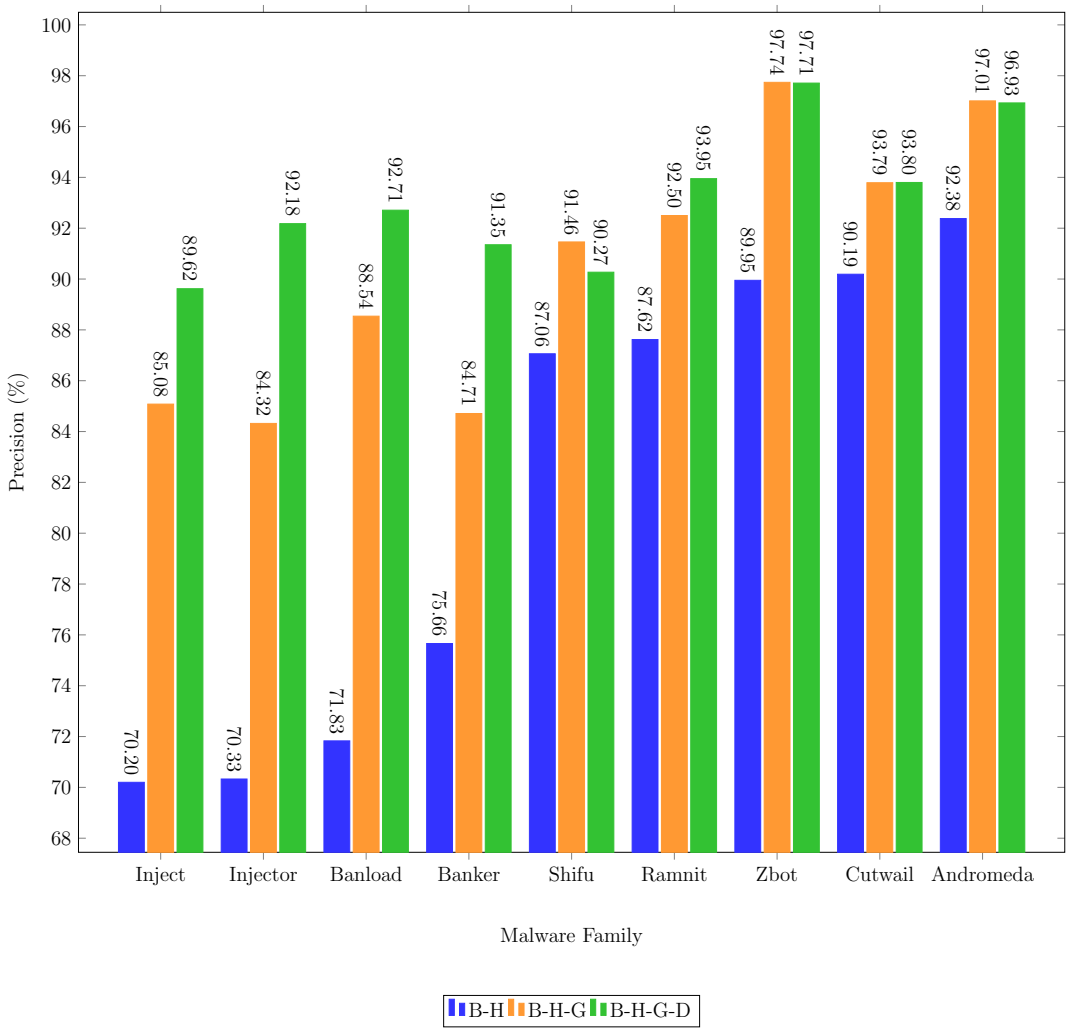

Figure 5.7: Individual precision results for malware classification models.

relevance of our model. For instance, SEEMA could potentially predict all samples as requiring cheap-to-compute static features and still achieve an accuracy rate of more than 80%. This model, however, would not be good at recognizing malware that can only be classified with the use of more expensive characterizations. As a result, it is imperative that this class imbalance be corrected before being fed to our models. To solve this problem, we decided to *undersample* the observations that are part of the B class. Undersampling is one of the most common techniques used for handling class imbalance [Phatak, 2018]. In this technique, the majority class is down-sampled

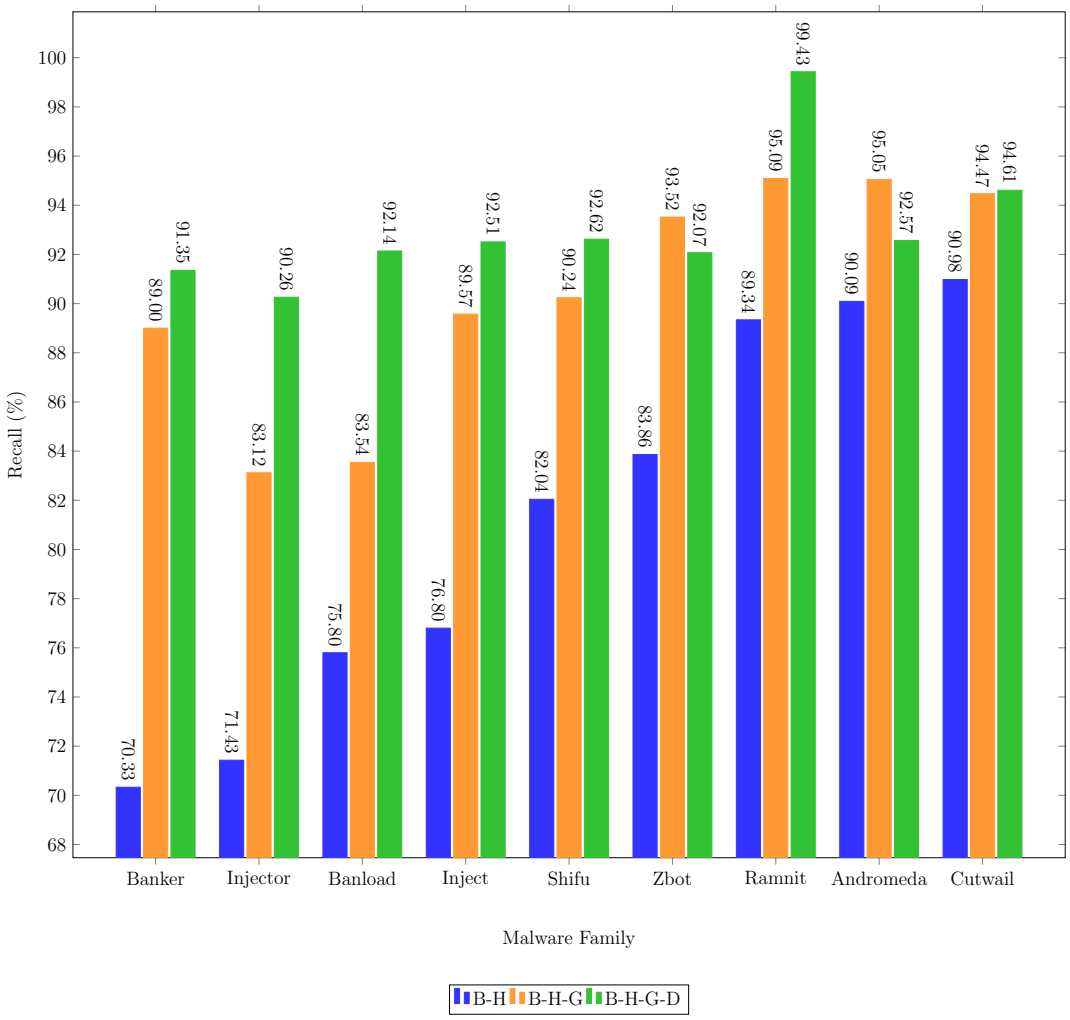

Figure 5.8: Individual recall results for malware classification models.

to match the number of observations of the minority class. Hence, we select random binaries with label B until we reach a total of 2477 (i.e., the number of samples whose label corresponds to B-G-D). The final composition of the input dataset is presented in Figure 5.9.

As previously shown in Figure 5.4, our SEEMA model was able to achieve an accuracy that is superior to the other three malware classification models (95.11% in contrast to 80.37%, 89.97%, and 93.02%, for the B, B-G, and B-G-D models, respectively). In terms of precision and recall, our surrogate model also managed to yield very high scores.

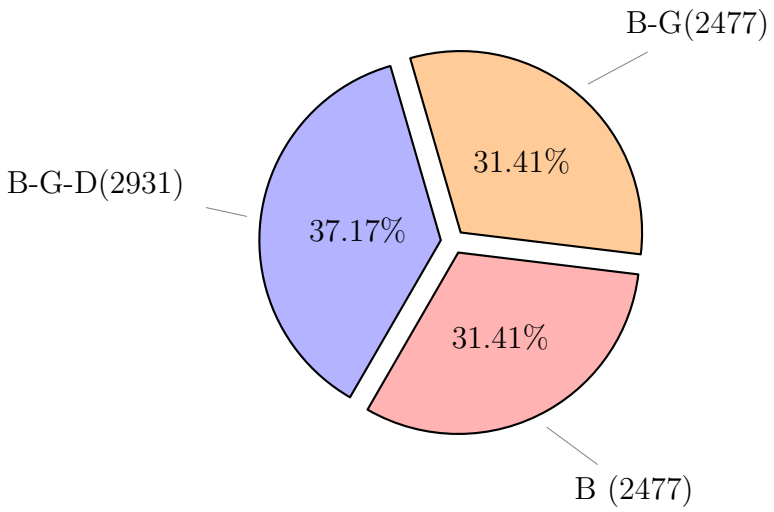

Figure 5.9: This figure illustrates the composition of the input dataset for our SEEMA model.

To further expand on these results, we show the confusion matrix for our SEEMA model as well as precision and recall results per class label. The large values on the diagonal of the matrix in Table 5.4 are indicative of the high predictive power of our SEEMA model.

		Predicted		
		B	B-G	B-G-D
Actual	B	**561**	23	2
	B-G	10	**467**	19
	B-G-D	6	17	**472**

Table 5.4: Confusion matrix for SEEMA model.

Furthermore, the recall results presented in Table 5.5 highlight the completeness of this classifier. SEEMA can return most of the relevant results and yield a low number of false negatives for the samples in our dataset. For instance, out of all the malware binaries that should have been predicted as needing the B model to be correctly classified, SEEMA was able to capture 95.73% of them. In addition, our classifier was able to identify 95.35% of the instances that should only have been categorized with the most expensive model (B-G-D).

	Target Label		
	B	B-G	B-G-D
Recall	95.73	94.15	95.35
Precision	97.22	92.11	95.74

Table 5.5: Precision and recall results for SEEMA model.

Finally, the output from our SEEMA model has the advantage of being highly interpretable because it provides an analyst with the probabilities that each of our classifiers will lead to a correct prediction for a specific sample, as illustrated in Table 5.6.

	Target Label		
	B	B-G	B-G-D
Probability	**89%**	9%	2%

Table 5.6: Output from logistic regression model.

5.7 Discussion

In this chapter, we aimed to answer the following two questions: How *effective* is fast static analysis as compared to slow dynamic analysis for the classification of malware? How *efficient* is our approach in terms of generating features for a large malware dataset? We address the first question by comparing the results obtained from our malware classification models. Our findings show that we can indeed use cheap-to-compute static characterizations to accurately assign malware into their corresponding families. In fact, the results for our B model demonstrate that simple byte features hold enough predictive power to accurately classify around 80% of the samples in our dataset, and only 20% require more expensive characterizations. This a crucial finding because it allows us to determine when more computationally-intensive analyses improve the classification of malware.

These results are further enhanced by the ability of our surrogate model to select the simplest static characterizations that lead to an accurate classification of malware. Our results show that SEEMA produces high true positive rates (i.e., recall)

in terms of identifying malware that require our B and B-G-D models for an accurate classification. This is a vital factor for our analysis system as long periods of time will not be spent analyzing samples that only require simple characterizations to be correctly classified. Similarly, SEEMA also identifies the binaries that demand more computational resources from our analysis system (i.e., files labeled as B-G-D). This enables us to decrease the overall time it takes to generate features for a large malware corpus.

5.7.1 Efficiency of SEEMA

We also computed the time saved by using our SEEMA model by comparing it to an approach for malware family prediction that relies exclusively on expensive dynamic characterization of malware. For the purpose of this comparison, we used the target labels predicted by SEEMA for the test dataset used to evaluate its performance. Additionally, we defined the following two methods for feature generation:

1. **Using dynamic analysis for all available samples:** the entire malware dataset is analyzed in a sandbox for over 300 seconds in order to generate dynamic-based features.
2. **Using the predictions made by our SEEMA model:** for each sample in the test dataset, we extract features according to the labels predicted by SEEMA. If the predicted label does not lead to an accurate classification, we extract the next possible characterization until the malware is correctly assigned into its corresponding family.

As shown in Figure 5.10, SEEMA requires the least amount of time to generate features for our test set. Specifically, it reduces the average time required for feature engineering by over 64%, compared to a traditional approach, in which all the samples are submitted for analysis in a sandbox environment for 5 minutes. As a result, we can leverage SEEMA to predict the simplest features and classifiers that work best for different malware datasets. This will allow us to scale malware analysis to upwards of 1 million samples, as we will only need to generate the most expensive dynamic characterizations for a small subset of the entire malware corpus.

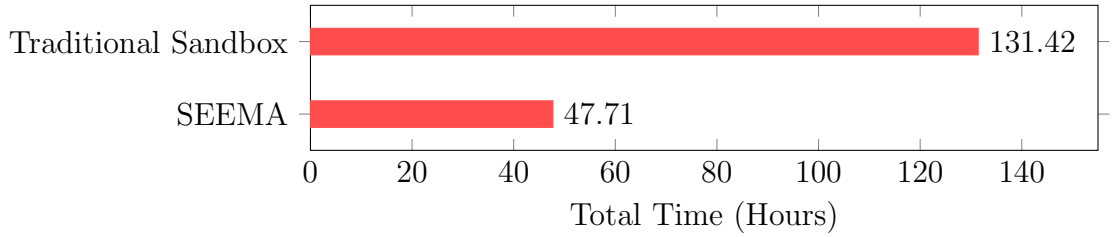

Figure 5.10: This figure depicts the time saved by using our SEEMA model compared to a traditional dynamic analysis approach where malware is executed for 300 seconds. SEEMA requires the least amount of time (47.71 hours) and reduces the average execution time that is required to uncover the malicious intent of our test dataset by 84.01 hours or 63.92%.

5.8 Related Work

In this section, we review the related work on feature set selection for malware classification as well as the effect of using hybrid features (i.e., adding dynamic analysis to static information extracted from malicious code) for malware family prediction.

5.8.1 Feature Exploration

There are a few related works in the area of feature set construction for malware family classification. Moonsamy et al. [Moonsamy et al., 2011] investigated feature reduction to speed up malware classification. In their research, the authors leveraged information gain as a method to discriminate between important and less important string features extracted from malicious code. They hypothesized that the frequency of a string in a feature set is indicative of its importance to the malware classification process. By using this approach on a dataset of approximately 1824 executable samples, the authors managed to distinguish malware families with an accuracy of over 94.5% [Moonsamy et al., 2011].

Ahmadite et al. [Ahmadi et al., 2016] performed feature selection on a set of static features extracted from over 20,000 malware. Their feature space included metadata, byte-entropy histograms, byte-sequences, and api call frequency. By leveraging a parallel implementation of the gradient boosting tree classifier, they succeeded

in selecting the best combination of features for malware classification, attaining an accuracy rate of over 99% [Ahmadi et al., 2016].

Islam et al. [Islam et al., 2013] developed a model that identified the best combination of function and string features for malware classification. The authors hypothesized that a classifier trained on these two characterizations would achieve better results than if the two characterizations were done separately due to their independent nature. Their approach was tested on a dataset of over 1400 malware samples using five different classifiers, with random forest yielding the best performance in terms of accuracy classification (98%) [Islam et al., 2013].

On the other hand, Vadrevu and Perdisci [Vadrevu and Perdisci, 2016] attempted to reduce the need for dynamic-based features by identifying malware that could be detected using fast static analysis information. In their work, the authors used static-based features to learn behavioral profiles from a training dataset. These attributes included number and size of memory sections as well as byte-entropy features. If the static features of a new malware sample perfectly matched those of a learned malware family, it was determined that dynamic analysis was not required for the file in question. This approach, however, has a significant limitation as its similarity function only allows it to filter out binaries with identical static analysis features [Vadrevu and Perdisci, 2016].

The previous approaches focus on determining the particular combination of features that lead to the highest accuracy rate for the problem of malware classification. However, in our research, we do not only integrate different types of features into a highly accurate malware classification model, but we also determine which particular classifiers and features work best for *each individual sample* in our dataset. As demonstrated in our results section, simple and fast static characterization of malware, such as bytes features, hold enough predictive power to accurately identify malware, and more expensive characterizations, such as assembly or dynamic features, are not always required. This allows us to increase the throughput of our system as expensive features are only necessary for a small portion of our malware corpus.

5.8.1.1 Hybrid Characterizations of Malware

Approaches leveraging both static and dynamic information in the context of malware classification have been proposed by numerous authors [Robiah et al., 2009], [Santos et al., 2013], [Hu and Shin, 2013], [Eskandari et al., 2013],[Elhadi et al., 2012], [Choi et al., 2012], and [Damodaran et al., 2017]. The general consensus is that hybrid characterizations of malware generally improve the performance of machine learning classifiers compared to models built exclusively on static or dynamic features. Some of the proposed approaches are further described in the following sections.

Santos et al. [Santos et al., 2013] examined the benefits of adding dynamic information to predictive features based on static analysis information. For their hybrid approach, the authors used *opcode-sequences* (i.e., instructions that specify the action to perform in machine code language), which was extracted from static analysis, as well as system calls, operations, and exceptions aggregated during the execution of malware in an emulated environment. In addition, this feature space was reduced by means of information gain and combined to form a hybrid static-dynamic input dataset. This approach was tested on a set of 2000 executables using four machine learning models that included k-nearest neighbors, decision tree, support vector machine, and Bayesian network algorithms. Their results showed that their hybrid approach outperformed the results of models trained exclusively on static or dynamic information from malware, with an average accuracy of over 96% [Santos et al., 2013].

Xin and Shin [Hu and Shin, 2013] combined information from both dynamic and static analyses to cluster malware families. Static information included opcodes, whereas dynamic features were constructed based on sequences of API calls collected during the execution of malware in a sandbox environment. Using a clustering ensemble method, the authors tested the effectiveness of their approach on a dataset of over 5647 malware files. Their results showed that a model trained on a hybrid characterization of malware could generate a better accuracy score for the classification of malware (98.72%), as opposed to models built on a specific set of features extracted from malicious code, such as static analysis (84.9%) or dynamic analysis (87.5%) [Hu

and Shin, 2013].

Eskandari et al. [Eskandari et al., 2013] developed a tool called HDM Analyzer that takes advantage of both static and dynamic information during the training portion of their algorithms but uses only static analysis in the testing stage. The authors intended to leverage the high predictive power of dynamic analysis information in the training phase while exploiting the efficiency of fast static analysis for the testing stage, a concept that we also use in this dissertation. Their results showed that their hybrid approach had a higher overall accuracy (95.27%) than a model built on static analysis features alone (89.43%) [Eskandari et al., 2013].

Our models significantly extend the previous approaches by creating a method that determines when the classification of malware benefits from hybrid features. Although it has been demonstrated that combining dynamic and static information generally leads to more accurate models, cheap-to-compute static features may still suffice to classify a significant number of malicious binaries. As a result, dynamic analysis is not always required and a hybrid approach might not necessarily yield a better prediction. Our SEEMA model leverages this concept and determines when dynamic analysis is actually required to improve the classification of malware beyond performing simple static analysis. As a result, we need to generate the most expensive characterizations for only a small subset of a large malware corpus, which allows us to scale malware analysis to large datasets of over 1 million malicious samples.

5.9 Conclusions

This chapter introduced a series of models of increasing cost and predictive power for the classification of malware. These classifiers are constructed based on information extracted from both static and dynamic analysis of malware, namely byte-level information, graph-based features, and dynamic analysis information. Our results showed that a hybrid approach (i.e., a model constructed by using all available static and dynamic features) produced the highest mean accuracy (93.02%) for malware classification. The model built using the simplest static characterization, however, was

also able to generate model results, with an average accuracy of over 80.37%. These results demonstrate that a significant portion of our malware dataset only requires cheap-to-compute features to be correctly assigned into their corresponding families.

The results from our malware classifiers were used to train a surrogate model, which we referred to as SEEMA (**S**electing the most **E**fficient and **E**ffect **M**alware **A**ttributes). SEEMA correctly predicted when less expensive malware characterization, such as static analysis, would suffice to accurately classify malware. This allowed us to *filter* the malware that would *not* benefit from time-consuming executions in an emulated environment. Given that in practice a large portion of malware does not require expensive-to-compute dynamic features, and since static analysis is generally orders of magnitude faster than dynamic analysis, we managed to significantly decrease the time spent extracting features for malware classification and to improve the throughput of our intelligent malware analysis system.

Chapter 6

MAGIC: MALWARE ANALYSIS TO GENERATE IMPORTANT CAPABILITIES

6.1 Introduction

Manually constructed malware analysis platforms that identify important capabilities in malicious software cannot keep up with the massive amounts of malware being released on a daily basis [Harrison and Pagliery, 2015b]. Traditional approaches that detect the functional capabilities of malware usually contain brittle handcrafted heuristics that quickly become outdated and can be exploited by nefarious actors. As a result, it is necessary to change the way software security is approached by using advanced analytics (i.e., machine learning) and significantly more automation to develop more adaptable malware analysis engines that correctly deduce the important capabilities of malware.

In this chapter, we introduce *MAGIC* (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities). MAGIC is machine learning-based model that learns from static and dynamic characterization of malware and automate malware capability discovery. It uses static features of malicious code and capabilities obtained from Reversing Labs (a malware dataset provider) and Cuckoo Sandbox, to map fast static analysis characteristics that indicate malicious behaviors. By using static characterization of our malware dataset, MAGIC can dramatically reduce the time that is required to deduce high-level functionalities of malware.

6.2 Problem Statement

One of the most challenging problems when analyzing malicious software corresponds to the identification of capabilities of malware. *Malware capabilities* are high-level functionalities of malicious software [Saxe et al., 2014], such as the ability to detect the presence of an analysis environment, capture key strokes or encrypt a user's personal information. For decades, security professionals developed techniques to analyze malware manually and deduce its capabilities [Landesman, 2017]. Malicious applications were smaller and less complex, and it was simpler to perform manual inspection to both detect malware and infer its important capabilities.

However, malware has become increasingly complex and numerous [Huang and Stokes, 2016], demanding adaptive-based techniques for inferring the functionality of malicious code. Furthermore, unlike data for problems such as the malware family prediction, which can be retrieved from most antivirus vendors, information regarding the capabilities of malware binaries is not as readily available [Saxe et al., 2013]. An explanation for this phenomenon is found in the research conducted by Saxe et al. [Saxe et al., 2014]. In their work, the authors express that semantic representation of high-level features of the malware has traditionally required extensive knowledge engineering work. Standard formats, such as YARA [Alvarez, 2017], that are used for describing malware, are constructed using static and dynamic features extracted from malicious code and rely entirely on a large number of rules that are inherently brittle and difficult to maintain.

As described in Chapter 2, machine learning has recently emerged as a state-of-the-art approach to automate the construction of heuristics used in malware analysis systems. When trained on meaningful characterizations of malicious code, machine learning techniques have proven to be much more resilient to noise than traditional handcrafted methods of analyzing malware and can yield better malware classification models [Kolosnjaji et al., 2016b].

Recently, it was demonstrated that machine learning models trained using features extracted from the dynamic execution of malicious code could be used to predict

behavioral components of malware, a problem related to the identification of malware capabilities [Yavvari et al., 2012]. By executing malicious software in an isolated sandbox environment, it is possible to extract knowledge from the malware and construct behavioral profiles that describe the functionality of the binary. Nevertheless, dynamic analysis can be a bottleneck for analyzing malware. As mentioned in Chapter 4 and Chapter 5, traditional sandbox environments take a long time to execute suspected malware, often 5 minutes or more [Damodaran et al., 2017]. This poses a challenge because dynamic analysis needs to be orders of magnitude faster to cope with the ever-increasing load of malware that appears daily.

Static analysis compensates for the computational expense of dynamic analysis since performing static analysis on malicious binaries can often be performed in seconds [Vadrevu and Perdisci, 2016]. As shown in Chapter 5, Figure 5.1, byte-level information takes less than a second to produce, even for significantly large malware files. More expressive characterizations of malware, such as graph-based features constructed from assembly code, usually take longer to generate, but can be extracted faster than information derived from slow dynamic analysis. As a result, machine learning classifiers trained on static features can lead to predictive models that are able to scale to large amounts of malware in a relatively short time.

6.3 Approach

We want to build a machine learning-based model that predicts important capabilities of malware. To accomplish this, we propose *MAGIC* (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities). MAGIC learns from static and dynamic characterizations of malware and automate malware capability discovery. Using static features of malicious code, as well as capabilities and indicators obtained from Reversing Labs and Cuckoo Sandbox, MAGIC can map fast static analysis characteristics that indicate malicious behaviors. In addition, by using static characterization of our malware dataset, MAGIC reduces the time that is required to infer high-level functionalities of malware.

To induce accurate machine learning models for predicting important capabilities of malware, we build a dataset using features and targets derived from both static and dynamic analysis.

6.3.1 Features

Bytes and hashes histograms can be calculated faster than most other features, even other static ones. However, we believe that these two characterizations of malware do not lead to overly intuitive or robust models. It is difficult to gain intuition from a file's byte entropy values and using strings in a file that can easily be changed will not lead to robust models. We therefore choose to focus on instruction-based features, which correspond to the code generated by the compiler used to create the binary.

6.3.1.1 Instruction-Based Features

We use Radare2 to translate the machine code into assembly code and generate instruction vectors for our malware dataset. First, the executable is disassembled into function, block, and operation level information, which corresponds to the executable's call graph, control flow graph, and instruction flow graph, respectively. At each level of assembly graph for the given executable, we extract adjacency lists, which hold the graph's node and edge information, and an instruction histogram for each node in the graph. Each instruction in the instruction histogram maps to one of the 53 categories listed in Table 6.1 [Searles et al., 2017]. We choose to use the block level representation of the executable to build our feature vectors because it is more manageable in size than the operation level and generally more expressive than the function level.

6.3.1.2 Global Instruction Features

A global feature vector representing all instructions in the binary is constructed in two different ways.

control flow	switch ucjmp	case uccall	call ccall	ucall ret	jmp cret	ujmp swi	cjmp not
arithmetic	length sub sal ror	cmp abs sar rol	acmp mul or	add div and	mod shr xor	cast shl crypto	cpl nor
memory	mov upush	lea pop	cmov push	xchg new	leave io	store	load
miscellaneous	null	nop	unk	trap	ill		

Table 6.1: This table shows the 53 categories of instructions extracted using Radare2. These instructions correspond to one of four major categories, namely: control flow, arithmetic, memory, and miscellaneous.

6.3.1.2.1 Global Instruction histograms

To generate global instruction histograms, we take every instruction histogram for each block in the control flow graph and sum them. This produces a complete histogram of instructions in the executable. This process is illustrated in Figure 6.1.

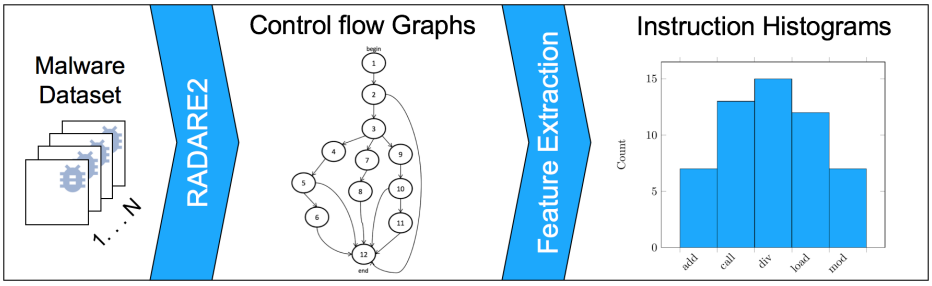

Figure 6.1: This figure illustrates the process of constructing instruction histograms from the block level representation (control flow graph) of a malicious binary. We sum the instruction histograms for every node in the graph, resulting in a global representation of the instructions used in the executable. We can also convert this histogram into a bit vector by changing all non-zero entries to a "1".

6.3.1.2.2 Global Instruction bit vectors

We also include a binary representation of the global instruction histogram. Any instruction that has a value of one or greater will be converted to a "1"; any instruction

that has a value of less than one will be onverted to a "0". This amounts to a bit vector of 53 entries representing where each of the 53 instructions appear in the binary.

6.3.1.3 Individual Node Features

We can also construct feature vectors from information about individual nodes in the graph, versus using a feature vector to express a summarized view of all the instruction in all the nodes in the graph. Feature vectors that pertain to individual nodes in the graph give more fine-grained information about the behavior of the program, and lead to more intuitive models. We use random walks of our graphs and construct feature vectors of the nodes that were walked. In this chapter, we performed a random walk method to generate individual node features.

6.3.1.3.1 Random Walk

We complete random walks of the block level representation of the executable to produce feature vectors. We extract adjacency lists that hold node and edge information of the control flow graph and an instruction histogram for each node in the graph. The entry point of the control flow graph is the chosen start position of our walk. For the length of the walk, a random choice is made from the node's neighbors and that node becomes the new start node. This iterates until the walk is completed. If a node has no neighbors to visit, we end the walk. We complete multiple walks in this fashion and output a list of visited nodes. The final feature vector for a random walk is compiled by concatenating the instruction histogram for each visited node. If the walk was ended early, we pad the feature vector with zero values for its instruction histogram for each node that could not be visited. We empirically choose 2 random walks of length 5 for a total of 10 visited nodes. A depiction of this process is shown in Figure 6.2.

6.3.1.3.2 Random Walk bit vector

Similarly to the global instruction bit vectors, we include a binary representation of the random walk which is similar, but each instruction is either 0 if that instruction

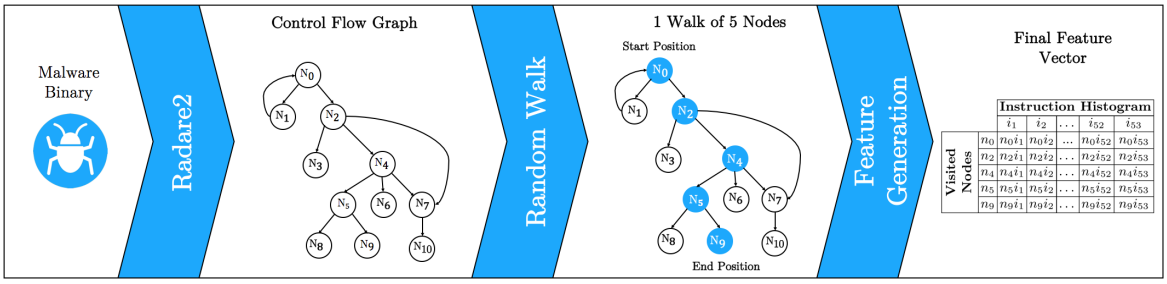

Figure 6.2: This figure shows the process to complete random walks of the block level representation of a malicious binary. For a given walk of 5 nodes, we use the entry point of the control flow graph as the start position of our walk. For the length of the walk, a random choice is made from the node's neighbors and that node becomes the new start node. This iterates until the walk is completed and the output corresponds to a list of visited nodes. The final feature vector for a random walk is constructed by concatenating the instruction histogram for each visited node.

is not used at the node or 1 if the instruction appears.

6.3.1.4 Dynamic Characterization

The second critical component of our research is to compute the target capabilities that we will be predicting with our machine learning algorithms. To accomplish this, we leveraged two different sources of information regarding high-level capabilities of malware: Cuckoo Sandbox and Reversing Labs' A1000 Cloud Analysis System. We use these two analysis sources to generate two different sets of target capabilities. Using two different sources of capabilities allowed us to evaluate the power of our models in predicting capabilities, regardless of the system we used to generate our dataset.

6.3.1.5 Cuckoo Sandbox

The developers of Cuckoo have a-priori determined a set of behaviors that can be deemed as malicious, such as a file installing itself for autorun during Windows startup, or querying information about the computer, e.g., its uptime or browser history. When a malicious sample is executed in the sandbox environment, it is possible to run these behaviors against the analysis results and identify capabilities or indicators of interest. These behaviors provide a context to the analyses, as they simplify the interpretation

of the results and can help identify particular malware functionalities, such as anti-sandbox behavior. For the work proposed in this chapter, we explored this behavioral space to induce models that serve a proxy to running much more expensive dynamic analysis to predict the important capabilities of the samples in our malware dataset.

6.3.1.6 Malware Attribute Enumeration and Characterization

After a file is dynamically analyzed, Cuckoo generates a report that outlines malware indicators of interest. One of the reporting formats that Cuckoo offers corresponds to the **M**alware **A**ttribute **E**numeration and **C**haracterization format (MAEC). MAEC is a standardized file information sharing format that is used to describe the important capabilities of malware [Oktavianto and Muhardianto, 2013]. It reduces the inherent ambiguity and discrepancies of malware descriptions by providing a general reporting framework with a unified structure and vocabulary. This allows organizations and cyber analysts to have an unambiguous communication channel to describe the nature and attributes of malware.

6.3.1.7 A1000 Cloud Analysis System

Reversing Labs' A1000 Cloud System is a malware analysis appliance that integrates static analysis and a file reputation database to enable the extraction of indicators of interest [ReversingLabs, 2018]. There are many indicators that are common in malicious applications, like accessing and modifying registries, or writing to files in system directories. These behaviors indicate what the suspected sample is capable of doing, i.e., its capabilities, and are organized in priority order [ReversingLabs, 2018].

6.4 Learning Methodology

Our MAGIC model automates the search for the most expressive methods of characterizing malware and leverages machine learning techniques to accurately identify important capabilities of malware. The architecture to generate this prediction model is shown in Figure 6.3.

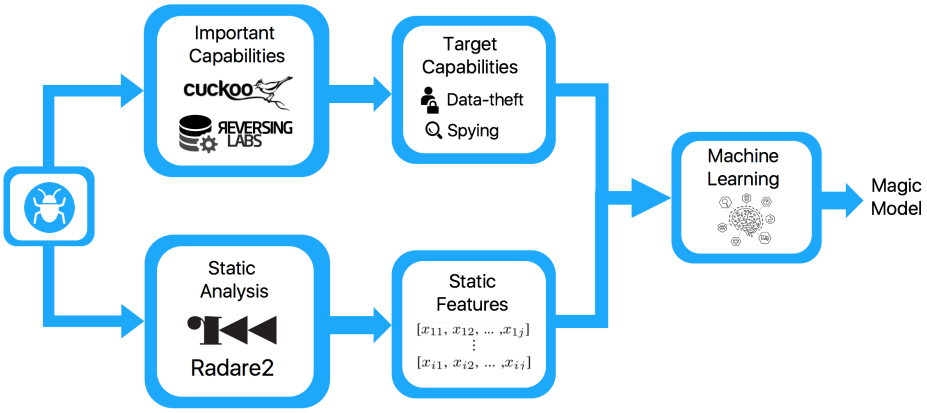

Figure 6.3: This figure illustrates the process to construct our MAGIC model. Static features are extracted using Radare2 and malware capabilities are generated using Cuckoo and Reversing Labs' A1000. Static features are used as input to train machine learning algorithms that predict the important capabilities of malware.

The process to build our MAGIC model involves using Radare2 to extract static analysis information from each malware that we analyzed. In particular, we characterize malware using various different types of instruction features to form the input dataset of our model. The remaining part of our dataset includes the targets or labels for each malware required to train our classifiers. These target classes correspond to the important capabilities of the malware binaries and are extracted using two different analysis systems: Cuckoo Sandbox and Reversing Labs' A1000 Cloud Analysis System.

A total of 30073 files were analyzed using Cuckoo and over 59627 malicious executables were examined using the A1000. The reason for the discrepancy in dataset size stems from the fact that performing dynamic analysis in Cuckoo is a much more resource-intensive and time-consuming process. As a result, a smaller subset of malware was selected to be dynamically analyzed in our sandboxed environment.

We executed the samples for 1 minute and a report outlining their important capabilities was generated at the end of each run. Sixty seconds proved to be enough time to extract meaningful information from the majority of the samples in our dataset. However, some files did not display any malicious activity, even after the one-minute execution. Consequently, we reprocessed those files for an additional 4 minutes in order

to identify malware with potentially stalling code. On the other hand, the A1000 is a high-throughput malware analysis engine that can generate summary reports for each file submitted for scanning, and contains the indicators of interest of the suspected samples. However, while the A1000 scales better than Cuckoo, it is built using hand-crafted heuristics that become outdated as malware evolves.

Cuckoo was able to identify a total of seven malware capabilities across the 30073 files submitted for dynamic analysis, as indicated in Table 6.2.

Capability	Definition
Anti behavioral analysis	Refers to the ability of malware to detect the existence of an analysis environment, by searching for files and processes that are indicative of the presence of virtualization and emulation software [Assor, 2016].
Anti code analysis	Describes the presence of code designed to thwart static malware analysis, such as instructions to prevent disassembly and the generation of accurate call graphs [Kirillov et al., 2017].
Anti detection	Specifies the ability of malware to circumvent security software, by hiding processes that contain traces that indicate malicious behavior such as network traffic and registry artifacts [Kirillov et al., 2017].
Data theft	Defines the functionality of malware to compromise a user's personal files and information, such as images, documents, and authentication credentials.
Persistence	Designates the property of malware to remain on the infected system regardless of changes made by the user, such as an operating system reboot or reinstall.
Security degradation	Indicates that the suspected sample is able to deactivate the operating system's security features, such as security alerts and user account control.
Spying	Corresponds to the ability of malware to capture a user's keyboard, mouse or camera input, as well as information about the usage and browsing habits of the infected machine [Paloalto Networks, 2016].

Table 6.2: Cuckoo/MAEC Capabilities

The A1000 yielded a total of 22 indicators of interest (IOI), organized under seven categories, as presented in Table 6.3. These IOIs are akin to the capabilities we extracted from Cuckoo/MAEC.

Indicator	Descriptors
Evasion: tries to evade common debuggers/sandboxes/analysis tools	Uses anti-debugging methods
Monitor: monitors host activities	Detects/enumerates process modules Tampers with keyboard/mouse status
Execution: creates or starts processes	Writes to files in Windows system directories Creates/opens files in system directories Writes to files Reads from files Creates/Opens a file
File: accesses files in an unusual way	Executes a file Terminates a process/thread Might load additional DLLs and APIs Contains references to kernel32.dll Contains references to user32.dll
Settings: alters system settings	Enumerates system information Enumerates system variables
Registry: accesses registry and configuration files in an unusual way	Accesses-modifies registry
Search: collects system information	Checks operating system version Reads paths to system directories on Windows Enumerates files References executable file extensions References source code file extensions Enumerates user local information

Table 6.3: A1000 Indicators of Interest (IOI)

6.5 Experimental Infrastructure

Two different datasets were constructed, one using Cuckoo/MAEC capabilities and one using A1000 IOIs. The two datasets use the same features as inputs to the models that come from our static analysis data. We can use either Cuckoo or the A1000 to highlight the presence or absence of a discovered capability, so we construct two datasets, one for each of these analysis systems. A value of "1" is used to indicate the occurrence of a potentially malicious behavior in the execution of the file, whereas a value of "0" designates the lack of said behavior associated with the file. Table 6.4 presents a sample of target values constructed for one malware from the dataset that

was submitted for analysis in Cuckoo.

Malware (m)	Anti-Debugging	Anti-Sandbox	...	Data-Theft	Security-Degradation
56ed8678c0e1f...b5ab4c685dfd92	1	0	...	0	1

Table 6.4: Training instance for MAGIC model.

The static analysis information and a set of target values are then appended to yield an input training dataset. This information is then fed to the machine learning models to make the predictions for the important capabilities of malware.

6.5.1 Malware Family Distribution

We used the stream of malware that we obtained from Reversing Labs, which included malicious binaries targeting institutions in the financial services industry. Table 6.5 shows the name, type, and count of each family of our two datasets. Furthermore, our dataset was curated to guarantee the extraction of relevant information from the malicious files. As mentioned earlier, it is a well-known fact that most of the Windows malware are packed [Ugarte-Pedrero et al., 2015], and static analysis approaches can fail at extracting meaningful features from malware. Reversing Labs removed all packing, obfuscation, and protection artifacts from the binary files to extract all internal objects with their metadata. Hence, the unpacked malware were available for further analysis using our disassemblers.

6.5.2 Cuckoo/MAEC Capability Distribution

We generated evenly-distributed datasets for each Cuckoo/MAEC capability extracted from our malware executables. This was done to ensure that our machine learning models could be trained on a well-balanced dataset that contained both malware with and without a specific capability. For example, the dataset for the Cuckoo/MAEC data-theft capability included a total of 10543 samples, where 50% exhibited the aforementioned capability and the remaining 50% did not display said behavior during its

Name	Type	Count	
		Cuckoo/MAEC	A1000
Andromeda	virus	3516	4458
Banker	spyware	2730	8456
Banload	downloader	4358	8582
Cutwail	downloader	3018	4073
Inject	virus	3181	8193
Injector	trojan	2498	8522
Ramnit	trojan	3525	7169
Shifu	spyware	3497	4629
Zbot	downloader	3751	5545

Table 6.5: Distribution of Families for our Malware Datasets

execution. Finally, we created a total of seven datasets with even distributions for the Cuckoo/MAEC capabilities, as indicated in Table 6.6.

	Anti-behave	Anti-code	Anti-detect	Data-theft	Persis-tence	Security-deg	Spying
Total	9763	6740	22420	10543	20420	12272	9722

Table 6.6: Dataset distribution for Cuckoo/MAEC capabilities

6.5.3 A1000 Indicators of Interest Distribution

Our A1000 datasets is orders of magnitude larger than our Cuckoo/MAEC datasets. Therefore, we do not balance the A1000 datasets and instead use the data that we capture from our larger malware feed.

6.6 Experimental Results

After constructing the training dataset of relevant static features and important capabilities of malware, we used machine learning (ML) to induce prediction models. Specifically, we leveraged Sklearn [Pedregosa et al., 2011] to train our ML algorithms. In the following sections, we discuss the major results from the implementation of our MAGIC model using Sklearn. First, we discuss the datasets used to train, test, and evaluate our models. We then present preliminary results from different ML algorithms

evaluated on our malware datasets. Models trained using random forest achieved the highest average accuracy across all capabilities with a total value of 92.94%. Similar mean accuracies were obtained by decision trees (92.54%), and artificial neural networks (91.21%) across malware capabilities and indicators of interest. The least successful model in terms of average accuracy corresponded to support vector machine, which yielded a total value of 89.73%. For the remainder of the results, we focus on decision trees because they achieve performance results comparable to other ML algorithms and because their output models tend to be very intuitive. We explain the reasoning for the selection of this particular ML algorithm and show the advantages that this technique offers to cyber analysts in terms of automated malware capability discovery.

6.6.1 Decision Trees

Decision trees (DT) correspond to a machine learning technique that has the potential to be human interpretable and easy to visualize. This makes DTs advantageous to cyber security analysts trying to comprehend specific features embedded in the code of the malware that are responsible for its capabilities. Moreover, DT algorithms inherently perform useful tasks for human analysts, including feature importance and feature selection on a malware dataset, as the top few nodes on which the tree is split correspond to the most important features of the malware, upon which its important capabilities are derived [Deshpande, 2011b]. Finally, DTs do not require a considerable amount of computation to construct their models. In fact, our models built using DTs took on average around one minute to be trained and validated. We now present and discuss the results obtained after evaluating our proposed approach using DTs.

6.6.1.1 Decision Tree Results

In this section, we discuss the results generated by our decision tree classifier when trained on our smaller dataset of 30073 malware samples constructed using the Cuckoo/MAEC capabilities, as well as the larger dataset of 59627 malicious files built using the A1000. As shown in Figure 6.4 and Figure 6.5, our DT classifier models are

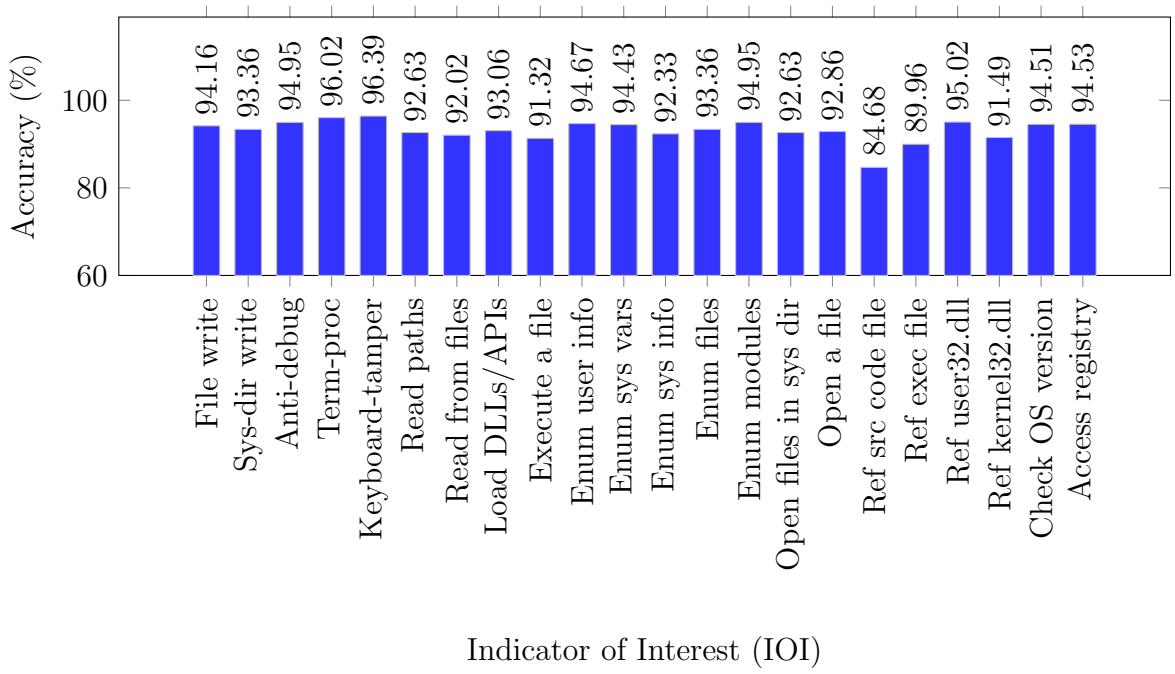

Figure 6.4: This figure shows accuracy results for the DT model evaluated using the A1000 IOIs

able to accurately predict the important capabilities of malware, achieving up to a 97% accuracy on predicting capabilities.

For our dataset constructed using the A1000 indicators of interest, the accuracy results were evenly distributed across capabilities. In fact, the results obtained for the prediction of twenty indicators of interest crossed the 90% threshold, which indicates the high predictive power of our models trained on instruction-based features. These particular results also highlight the high accuracy results that can be obtained when our models are trained using a large corpus of malware. The A1000 dataset contained more samples both with and without the 22 indicators of interest and, as a result, we were able to generate higher accuracies in contrast to the models trained on the smaller Cuckoo/MAEC dataset.

For our dataset constructed using Cuckoo/MAEC capabilities, the best performing results were obtained for the anti-detect capability, which achieved an accuracy of up to 97.11%, followed by security degradation (93.58%), data-theft (90.39%) and

persistence (90.03%). These results demonstrate the high predictive power of instruction histograms extracted from the code of malicious executables. This extremely fast static characterization holds enough information to accurately predict the important capabilities of malware.

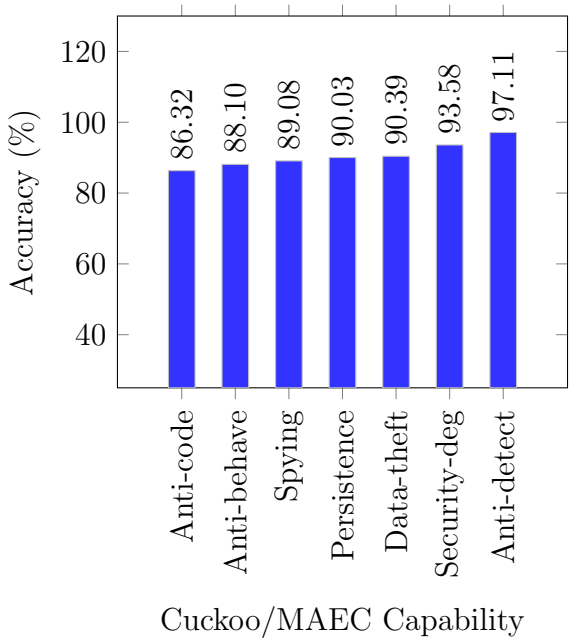

Figure 6.5: This figure shows accuracy results for the DT model evaluated using the Cuckoo/MAEC capabilities.

The worst performing capability corresponded to anti-code. Although the maximum accuracy obtained for this capability (86.32%) did not match the highly accurate results obtained for the other Cuckoo/MAEC capabilities, we still managed to make significant progress towards the discovery of malware that is likely to contain code designed to hamper static analysis, giving us the opportunity to focus on examining samples that do improve the intelligence of our analysis system.

6.7 Discussion

The results for our MAGIC model show that by using instruction-based features extracted from fast static analysis, we can accurately predict the important capabilities

of malware. Particularly, by using simple and inexpensive disassembly characterization of malware, such as instruction histograms, we are successful in removing the need to perform dynamic malware analysis to identify the important capabilities of malware. This enables us to drastically reduce the overall time that it takes to estimate the important capabilities of a large malware dataset because it is no longer necessary to run malware in a sandboxed environment for long periods of time in order to identify a malware's important capabilities.

Additionally, our prediction model makes significant progress towards the identification of crucial functionalities of malware such as anti-sandbox capabilities. By leveraging these predictions, it is possible to isolate malware that might contain logic to circumvent dynamic analysis if it detects the presence of an emulator or virtualized environment. This particular limitation can be overcome by running this subset of anti-sandbox malware on *bare-metal* architectures. Bare-metal systems provide a platform with no virtualization or emulation instrumentations, which allows analysts to observe the behavior of malware that otherwise would enable its anti-sandbox capabilities to circumvent analysis. Although bare-metal systems often incur high hardware costs and suffer from scalability limitations, our MAGIC model can effectively single out the anti-sandbox malware that will display its malicious behavior if executed on a physical machine.

Also, by predicting anti-detection capabilities of malware, our model determines which files are more likely to contain code designed to thwart security software by hiding processes that are indicative of malicious behavior. This gives us the opportunity to skip expensive dynamic analysis on samples that do not generate new information and thus can focus only on analyzing unseen malware that truly requires additional dynamic examination, drastically increasing the throughput of our malware analysis system.

6.7.1 Decision Tree Model

A DT constructed for the Cuckoo/MAEC persistence capability is shown in Figure 6.6. DTs can generate a visual model that can give security analysts more intuition about the capabilities of malware. This is accomplished by exploiting the benefits of instruction vectors extracted from the compiler representation graphs of malicious binaries. This particular characterization of malware provides analysts with a more readable description of the elements of malicious code associated with a specific capability. As seen in Figure 6.6, instruction-based DT models can highlight the most relevant instructions of the malware, upon which its important capabilities are derived.

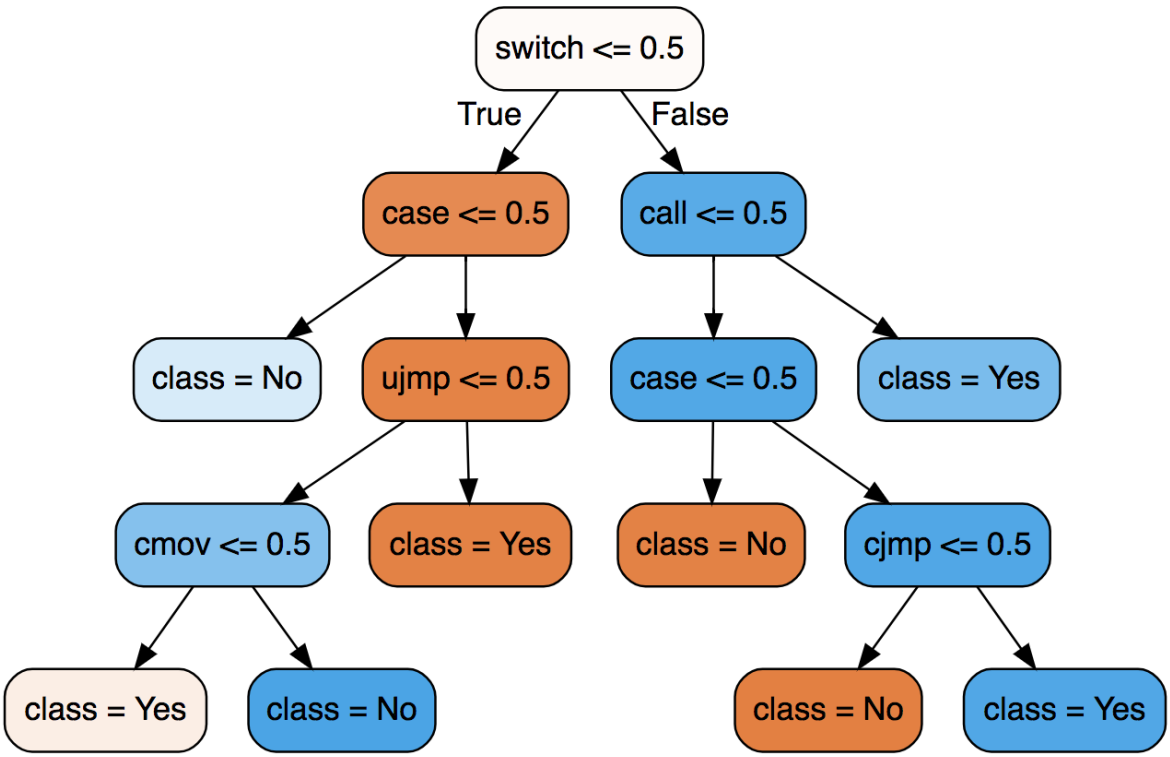

Figure 6.6: This figure depicts a DT constructed using instruction histograms derived from static analysis of malicious code. The information contained in the nodes can be used to predict a specific capability based on the presence (i.e., a node value greater than 0.5) or absence (i.e., a node value less than or equal to 0.5) of particular instructions in the code of the binary. For instance, the far-right nodes of the graph indicate that the existence of instructions such as switch and call can lead to the discovery of the anti-code analysis capability in the code of the malware.

In addition, we can use the information contained in the nodes to predict a specific capability based on the presence (i.e., if a node value is greater than 0.5) or absence (i.e., if a node value is less than or equal to 0.5) of particular instructions in the code of the binary. For instance, the far-right nodes of the graph indicate that the existence of instructions such as switch and call can lead to the discovery of the anti-code analysis capability in the code of the malware.

6.8 Related Work

Although the literature on important capabilities of malware is scarce, there a few related works in the area of malware functionality discovery.

Saxe et al. proposed using natural language processing (NLP) to identify important capabilities of malware and showed preliminary results regarding the feasibility of this approach in [Saxe et al., 2013]. A full implementation of this methodology was later conducted by the same authors in [Saxe et al., 2014]. First, they extracted function call symbols from the import table of the Portable Executable (PE) header of their malware corpus. In addition, they used dynamic analysis to retrieve information regarding the execution paths and registry modifications performed by the malware. The function call symbols embedded in their hybrid dataset were mapped to a corpus of 6 million posts mined from the website *www.stackoverflow.com* [Saxe et al., 2014]. Finally, the authors identified the occurrence rate of function call symbols within the posts and derived information about the important capabilities of the suspected samples.

On the other hand, there is a small number of commercial solutions such as JoeSandbox [JoeSandbox, 2017] and ThreatExpert [ThreatExpert, 2017], that provide an overview of the behavior of malware in an isolated environment and extract indicators of interest. For instance, JoeSandbox summarizes the network activities performed by the malware (e.g., downloaded files, DNS lookups) as well as file activities (e.g., number of created and deleted files) and registry activities as part of the report that it generates after the execution of the file. This behavioral space can be used to describe

and predict the capabilities that the malware exhibits during its execution. However, JoeSandbox is a commercial implementation of a dynamic analysis system and is therefore hampered by long analysis times and fragile human-constructed heuristics [JoeSandbox, 2017].

The open-source project YARA is another effort that tries to predict important capabilities of malware. YARA provides a framework to generate descriptions of malware families based on textual or binary patterns [Alvarez, 2017]. Typically, YARA rules consist of a set of *strings*, which correspond to descriptions of patterns-based malware families, and a Boolean expression to determine its logic. After a malicious sample is executed in a sandbox environment, it is possible to run these rules against the extracted behaviors and identify patterns or indicators of interest [Alvarez, 2017].

6.9 Conclusions

In this chapter, we introduced MAGIC (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities). First, we developed highly expressive and inexpensive static characterizations of malicious code to feed as input to our machine learning model. Specifically, MAGIC was constructed using input from instruction feature vectors that come from graphs of malicious code. Then, we extracted meaningful functionality information from our malware dataset, which was used as target labels for our machine learning algorithms. Our experimental results based on the decision tree classifier showed that by turning the malware capability prediction problem into a machine learning problem, it is possible to induce predictive models that estimate the high-level functionalities of malware with an accuracy of up to 97.11%. In addition, we demonstrated that by leveraging DTs it is possible to generate a visual model that is highly intuitive and can provide security analysts with insights about the specific features embedded in the code of the malware that are responsible for its capabilities.

Chapter 7

CONCLUSIONS

In this dissertation, we proposed the development of an intelligent sandbox environment that combines machine learning and static and dynamic analysis features, for the analysis and classification of malware. This combination of analyses attempts to cover their individual limitations, and produce models that are far more accurate and robust than using just one of the analysis techniques to produce a predictive model. Moreover, our machine-learning based models were able to address several important sandbox problems currently being solved by manually-tuned heuristics, such as 1) predicting the length of time needed to execute a malware, 2) learning when dynamic behavior improves the accuracy of malware detection beyond performing static analysis, and 3) automatically identifying important capabilities of malware.

In Chapter 4, we introduced a model that we refered to as TURACO (**T**raining **U**sing **R**untime **A**nalysis from **C**uckoo **O**utputs) that significantly extends the capabilities of Cuckoo, an open-source sandbox. TURACO leverages features extracted from *fast* static analysis of malware, as well as reports generated by Cuckoo that provide insights about the time needed to accurately classify a file as malicious. Our results demonstrated that TURACO could accurately identify the amount of time that malware should be executed for in a sandbox in order to reveal itself. This execution did not require a significant amount of time, as indicated by the subset of files that was deemed as malware in just under ten seconds. Additionally, we showed that it was possible to use static features to construct highly accurate machine learning models that estimate malware analysis run time. Furthermore, TURACO drastically reduced the average execution time required to uncover the malicious intent of a malware test dataset, compared to a method where all the samples were submitted for a total of

five minutes, which is the approach that traditional sandboxes use to examine malware behavior. Thusly, TURACO can be used to design and provision a *bare-metal* architecture (i.e., a system running on real hardware) for the dynamic analysis of malware at a lower cost.

In Chapter 5 we introduced a series of models of increasing cost and predictive power for the problem of malware classification. These classifiers were constructed based on information extracted from both static and dynamic analysis of malware, specifically: byte-level information, graph-based features, and dynamic analysis information. Our results showed that a hybrid approach (i.e., a model constructed using all available static and dynamic features) produced the highest mean accuracy for malware classification. Nevertheless, the model built using the simplest static characterization also managed to generate modest results, which indicated that a significant portion of our malware dataset only required cheap-to-compute features to be correctly assigned into their corresponding families. The results from these classifiers were used to train a surrogate model, that we referred to as SEEMA (**S**electing the most **E**fficient and **E**ffect **M**alware **A**ttributes). SEEMA correctly predicted when less expensive malware characterization, such as static analysis, would suffice to accurately classify malware. This allowed us to *filter* the malware that would *not* benefit from time-consuming executions in an emulated environment. Given that in practice a large portion of malware does not require expensive-to-compute dynamic features, and since static analysis is generally orders of magnitude faster than dynamic analysis, we managed to significantly decrease the time spent extracting features for malware classification, and improve the throughput of our intelligent malware analysis system.

Finally, in Chapter 6 we introduced MAGIC (**M**alware **A**nalysis to **G**enerate **I**mportant **C**apabilities). First, we developed highly expressive and inexpensive static characterizations of malicious code to feed as input to our machine learning model. Specifically, MAGIC was constructed using input from instruction feature vectors that come from graphs of malicious code. Then, we extracted meaningful functionality information from our malware dataset, that was used as target labels for our machine

learning algorithms. Our experimental results based on the decision tree classifier showed that, by turning the malware capability prediction problem into a machine learning problem, it is possible to induce predictive models that estimate the high-level functionalities of malware. In addition, we demonstrated that by leveraging decision trees it is possible to generate a visual model that is highly intuitive, and can provide security analysts with insights about the specific features embedded in the code of the malware that are responsible for its capabilities.

Chapter 8

FUTURE WORK

In the future, we plan to test the effectiveness of our models on a larger dataset that contains different types of malware. One of the limitations of the dataset used in this dissertation is that it only included windows executables. However, this malware corpus can be expanded to handle any kind of malware. As long as the dynamic analysis generates a report for a malware, that instance can be included in our model. Furthermore, we only examined malware using MS Windows 7 Professional, as it was compatible with our bare-metal and cloud-computing environments. Using different operating systems (OS) for the sandbox environment will give us the ability to detect maliciousness in malware that are built specifically to attack other OS architectures (e.g., Linux, iOS, and Android).

Additionally, some malware have the ability to detect the presence of a monitored environment and stall the execution of any malicious activity. As a result, we plan to investigate anti-anti sandbox capabilities of malware. Anti-sandboxing refers to the capability of malware to shut down or switch to benign behavior if the malware detects it is running in a sandboxed environment [Kirat et al., 2014]. Some anti-sandbox techniques performed by malware include checking the uptime of analysis system, or simply verifying whether sandbox software is installed and running on the computer [Assor, 2016]. We intend to invent new techniques to bypass anti-sandbox code that malware uses. This will be an important leap forward in the way sandbox environments are designed and developed. We will extend Cuckoo with the ability to automatically identify anti-sandbox capabilities. We will develop machine learning techniques to predict the code in malware that is anti-sandbox, and that can be safely ignored because it is being used to avoid sandbox introspection.

Accurately identifying the anti-sandbox statements that are placed in a program and that can be safely ignored will be a challenging problem. However, this is a problem well suited to Reinforcement Learning (RL), a machine learning technique that uses positive and negative rewards to train its models [Kaelbling et al., 1996]. Reinforcement learning corresponds to a type of machine learning algorithm in which the learning system, or *agent*, can observe an *environment* and perform *actions* based on *feedback* from the environment [Champandard, 2016]. The agent must then determine the ideal behavior or *policy* to maximize the rewards over time or minimize the penalties (negative rewards) received from the environment [Géron, 2017].Our RL models will predict code that is anti-sandbox and can be safely ignored. We will give positive rewards to our model based on the length of time a malware executes. If we skip code that causes a malware to crash or end its execution abruptly, we will give a negative reinforcement to our model to not skip that code. We will run thousands of independent executions of a malware in our sandboxed environment, and we are confident that over many executions we can develop highly accurate RL models that will accurately skip anti-sandbox code.

Moreover, we plan to explore additional static characterizations of malware to further increase the predictive power of MAGIC. Particularly, we plan to study other compiler representation graphs of malicious binaries to train MAGIC and improve our results for malware capability discovery. Finally, we intend to train and validate our approach using deep learning. Deep learning has recently emerged as the state-of-the-art technique for malware detection and classification, due to increasing computational capabilities and extensive datasets [Saxe and Berlin, 2015]. It has been proven that deep learning models can help analysts achieve breakthrough results in terms of both high accuracy, and low false positive rates for malware characterization and classification [Kolosnjaji et al., 2016b]. However, deep learning models are not inherently intuitive to analysts. We believe that analysts will have to decide whether to use deep learning based on a tradeoff of accuracy versus readability of the model.

BIBLIOGRAPHY

[Ahmadi et al., 2016] Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., and Giacinto, G. (2016). Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 183–194. ACM.

[AlAhmadi and Martinovic, 2018] AlAhmadi, B. A. and Martinovic, I. (2018). Malclassifier: Malware family classification using network flow sequence behaviour. In *Anti Phishing Working Group (APWG) Symposium on Electronic Crime Research (eCrime), 2018*, pages 1–13. IEEE.

[Albon, 2017] Albon, C. (2017). Nested cross validation. Retrieved from: https://chrisalbon.com/machine_learning/model_evaluation/nested_cross_validation/.

[Alvarez, 2016] Alvarez, S. (2016). Android malware analysis with radare: Dissecting the triada trojan. Retrieved from: https://www.nowsecure.com/blog/2016/11/21/android-malware-analysis-radare-triada-trojan/.

[Alvarez, 2017] Alvarez, V. (2017). Yara: The pattern matching swiss knife for malware researchers. Retrieved from: https://virustotal.github.io/yara/.

[Anderson et al., 2011] Anderson, B., Quist, D., Neil, J., Storlie, C., and Lane, T. (2011). Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258.

[Anderson et al., 2012] Anderson, B., Storlie, C., and Lane, T. (2012). Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and Artificial Intelligence*, pages 3–14. ACM.

[Arora, 2016] Arora, T. (2016). File entropy in malware analysis. Retrieved from: https://www.talentcookie.com/2016/02/file-entropy-in-malware-analysis/.

[Assor, 2016] Assor, Y. (2016). Anti-vm and anti-sandbox explained. Retrieved from: https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/.

[AV-Test, 2016] AV-Test (2016). Security report 2015/16. Retrieved from: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf.

[Bailey et al., 2007] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J. (2007). Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 178–197. Springer.

[Bayer et al., 2009] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, volume 9, pages 8–11.

[Bayer et al., 2010] Bayer, U., Kirda, E., and Kruegel, C. (2010). Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1871–1878. ACM.

[Bierma et al., 2014] Bierma, M., Gustafson, E., Erickson, J., Fritz, D., and Choe, Y. R. (2014). Andlantis: Large-scale android dynamic analysis. In *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)*.

[Blasing et al., 2010] Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 55–62. IEEE.

[Brownlee, 2016] Brownlee, J. (2016). How to prepare your data for machine learning in python with scikit-learn. Retrieved from: https://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/.

[Cade, 2017] Cade, C. (2017). Understanding heuristic-based scanning vs. sandboxing. Retrieved from: https://www.opswat.com/blog/understanding-heuristic-based-scanning-vs-sandboxing.

[Cavazos, 2017] Cavazos, J. (2017). Malware analysis and detection using graph-based characterization and machine learning. US Patent App. 15/256,883.

[Champandard, 2016] Champandard, A. J. (2016). Reinforcement learning. Retrieved from: http://reinforcementlearning.ai-depot.com/.

[Choi et al., 2012] Choi, Y. H., Han, B. J., Bae, B. C., Oh, H. G., and Sohn, K. W. (2012). Toward extracting malware features for classification using static and dynamic analysis. In *8th International Conference on Computing and Networking Technology (ICCNT)*, pages 126–129. IEEE.

[Christodorescu and Jha, 2003] Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium*, volume 12, pages 12–12. USENIX Association.

[Christodorescu et al., 2008] Christodorescu, M., Jha, S., and Kruegel, C. (2008). Mining specifications of malicious behavior. In *Proceedings of the 1st India Software Engineering Conference*, pages 5–14. ACM.

[Cobb, 2016] Cobb, M. (2016). Why signature-based detection isn't enough for enterprises. Retrieved from: http://searchsecurity.techtarget.com/tip/Why-signature-based-detection-isnt-enough-for-enterprises.

[Costin et al., 2014] Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., and Antipolis, S. (2014). A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, pages 95–110.

[Cuckoo, 2017] Cuckoo (2017). https://cuckoosandbox.org/.

[Dahl et al., 2013] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3422–3426. IEEE.

[Damodaran et al., 2017] Damodaran, A., Di Troia, F., Visaggio, C. A., Austin, T. H., and Stamp, M. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12.

[Deshpande, 2011a] Deshpande, B. (2011a). 4 key advantages of using decision trees for predictive analytics. Retrieved from: http://www.simafore.com/blog/bid/62333/4-key-advantages-of-using-decision-trees-for-predictive-analytics.

[Deshpande, 2011b] Deshpande, B. (2011b). 4 key advantages of using decision trees for predictive analytics. Retrieved from: http://www.simafore.com/blog/bid/62333/4-key-advantages-of-using-decision-trees-for-predictive-analytics.

[Dwinnell, 2007] Dwinnell, W. (2007). Data mining in matlab. Retrieved from: http://matlabdatamining.blogspot.com/2007/02/stratified-sampling.html.

[Egele et al., 2012] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6.

[Elhadi et al., 2012] Elhadi, A. A., Maarof, M. A., and Osman, A. H. (2012). Malware detection based on hybrid signature behaviour application programming interface call graph. *American Journal of Applied Sciences*, 9(3):283.

[Eskandari et al., 2013] Eskandari, M., Khorshidpour, Z., and Hashemi, S. (2013). Hdm-analyser: a hybrid analysis approach based on data mining techniques for malware detection. *Journal of Computer Virology and Hacking Techniques*, 9(2):77–93.

[Firdausi et al., 2010] Firdausi, I., Erwin, A., Nugroho, A. S., et al. (2010). Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pages 201–203. IEEE.

[Fisher, 2017] Fisher, T. (2017). What is the windows registry? Retrieved from: https://www.lifewire.com/windows-registry-2625992.

[GData Internet Security, 2017] GData Internet Security (2017). History of malware. Retrieved from: https://www.gdata-software.com/seccuritylabs/information/history-of-malware.

[Géron, 2017] Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media Inc., Sebastopol, CA.

[Grosse et al., 2016] Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. (2016). Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*.

[Guo et al., 2008] Guo, F., Ferrie, P., and Chiueh, T.-C. (2008). A study of the packer problem and its solutions. In *Research in Attacks, Intrusions and Defenses (RAID)*, volume 8, pages 98–115. Springer.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter*, 11(1):10–18.

[Hansen et al., 2016] Hansen, S. S., Larsen, T. M. T., Stevanovic, M., and Pedersen, J. M. (2016). 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.

[Hardy et al., 2016] Hardy, W., Chen, L., Hou, S., Ye, Y., and Li, X. (2016). Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee

of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[Harrison and Pagliery, 2015a] Harrison, V. and Pagliery, J. (2015a). Nearly 1 million new malware threats released every day. Retrieved from: http://money.cnn.com/2015/04/14/technology/security/cyber-attack-hacks-security/index.html.

[Harrison and Pagliery, 2015b] Harrison, V. and Pagliery, J. (2015b). Nearly 1 million new malware threats released every day. Retrieved from: http://money.cnn.com/2015/04/14/technology/security/cyber-attack-hacks-security/.

[Holmes, 2014] Holmes, J. (2014). Knowledge discovery in biomedical data: theory and methods. *Methods in Biomedical Informatics A Pragmatic Approach*, pages 179–240.

[Honig and Sikorski, 2012] Honig, A. and Sikorski, M. (2012). Hashing: A fingerprint for malware. Retrieved from: https://www.safaribooksonline.com/library/view/practical-malware-analysis/9781593272906/ch02s02.html.

[Hope, 2017] Hope, C. (2017). Assembly language. Retrieved from: https://www.computerhope.com/jargon/a/al.htm.

[Hosmer, 2008] Hosmer, C. (2008). Polymorphic and metamorphic malware. Retrieved from: https://www.blackhat.com/presentations/bh-usa-08/Hosmer/BH_US_08_Hosmer_Polymorphic_Malware.pdf.

[Hu and Shin, 2013] Hu, X. and Shin, K. G. (2013). Duet: integration of dynamic and static analyses for malware clustering with cluster ensembles. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 79–88. ACM.

[Hu et al., 2013] Hu, X., Shin, K. G., Bhatkar, S., and Griffin, K. (2013). Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198.

[Huang and Stokes, 2016] Huang, W. and Stokes, J. W. (2016). Mtnet: a multi-task neural network for dynamic malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 399–418. Springer.

[Hubicka, 2003] Hubicka, J. (2003). Control flow graph. Retrieved from: http://www.ucw.cz/~hubicka/papers/proj/node18.html.

[InfoSec, 2018] InfoSec (2018). Malware analysis basics - part 1 static malware analysis. Retrieved from: https://resources.infosecinstitute.com/malware-analysis-basics-static-analysis/.

[Islam et al., 2013] Islam, R., Tian, R., Batten, L. M., and Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656.

[Issa, 2012] Issa, A. (2012). Anti-virtual machines and emulations. *Journal in Computer Virology*, 8(4):141–149.

[Jang et al., 2011] Jang, J., Brumley, D., and Venkataraman, S. (2011). Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 309–320. ACM.

[JoeSandbox, 2017] JoeSandbox (2017). Retrieved from: https://www.joesecurity.org/.

[Ka, 2016] Ka, M. (2016). Finding advanced malware using volatility. Retrieved from: https://eforensicsmag.com/finding-advanced-malware-using-volatility/.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[Karim et al., 2005] Karim, M. E., Walenstein, A., Lakhotia, A., and Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23.

[Kaspersky, 2018] Kaspersky (2018). Types of malware. Retrieved from: https://usa.kaspersky.com/resource-center/threats/malware-classifications.

[Khanse, 2014] Khanse, A. (2014). Evolution of malware. Retrieved from: http://www.thewindowsclub.com/evolution-of-malware-virus.

[Kilgallon et al., 2017] Kilgallon, S., De La Rosa, L., and Cavazos, J. (2017). Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. In *2017 Resilience Week (RWS)*, pages 1–6. IEEE.

[Kirat et al., 2011] Kirat, D., Vigna, G., and Kruegel, C. (2011). Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM.

[Kirat et al., 2014] Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security*, volume 2014, pages 287–301.

[Kirillov et al., 2017] Kirillov, I., Beck, D., Chase, P., and Martin, R. (2017). Maec 5.0 specification, vocabularies. Retrieved from: http://maecproject.github.io/releases/5.0/MAEC_Core_Specification.pdf.

[Kolosnjaji et al., 2016a] Kolosnjaji, B., Zarras, A., Lengyel, T., Webster, G., and Eckert, C. (2016a). Adaptive semantics-aware malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer.

[Kolosnjaji et al., 2016b] Kolosnjaji, B., Zarras, A., Webster, G., and Eckert, C. (2016b). Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer.

[Kolter and Maloof, 2004] Kolter, J. Z. and Maloof, M. A. (2004). Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM International Conference on Knowledge Discovery and Data Mining*, pages 470–478. ACM.

[Kong and Yan, 2013] Kong, D. and Yan, G. (2013). Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM International Conference on Knowledge Discovery and Data Mining*, pages 1357–1365. ACM.

[Kruegel et al., 2004] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. (2004). Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, volume 13, pages 1–18.

[Landesman, 2017] Landesman, M. (2017). A brief history of malware. Retrieved from: https://www.lifewire.com/brief-history-of-malware-153616.

[Lemonnier, 2015] Lemonnier, J. (2015). What is malware, how malware works and how to remove it. Retrieved from: https://www.avg.com/en/signal/what-is-malware.

[LMSecurity, 2017] LMSecurity (2017). Static malware analysis. Retrieved from: http://resources.infosecinstitute.com/malware-analysis-basics-static-analysis/.

[Lo et al., 1995] Lo, R. W., Levitt, K. N., and Olsson, R. A. (1995). Mcf: A malicious code filter. *Computers & Security*, 14(6):541–566.

[McGraw and Morrisett, 2000] McGraw, G. and Morrisett, G. (2000). Attacking malicious code: A report to the infosec research council. *IEEE software*, 17(5):33–41.

[Microsoft, 2010] Microsoft (2010). Metadata and the pe file structure. Retrieved from: https://msdn.microsoft.com/en-us/library/8dkk3ek4(v=vs.100).aspx.

[Microsoft, 2018] Microsoft (2018). Pe format. Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format#import_directory_table.

[Microsoft, 2018] Microsoft (2018). Trojandownloader:win32/banload. Retrieved from: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Banload.

[Moonsamy et al., 2011] Moonsamy, V., Tian, R., and Batten, L. (2011). Feature reduction to speed up malware classification. In *Nordic Conference on Secure IT Systems*, pages 176–188. Springer.

[Moser et al., 2007] Moser, A., Kruegel, C., and Kirda, E. (2007). Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245. IEEE.

[Mujumdar et al., 2013] Mujumdar, A., Masiwal, G., and Meshram, D. B. (2013). Analysis of signature-based and behavior-based anti-malware approaches. *International Journal of Advanced Research in Computer Engineering and Technology*, 2(6):2037–2039.

[Narayanan et al., 2013] Narayanan, A., Chen, Y., Pang, S., and Tao, B. (2013). The effects of different representations on static structure analysis of computer malware signatures. *The Scientific World Journal*, 2013(13):1–8.

[Nari and Ghorbani, 2013] Nari, S. and Ghorbani, A. A. (2013). Automated malware classification based on network behavior. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 642–647.

[Nath and Mehtre, 2014] Nath, H. V. and Mehtre, B. M. (2014). Static malware analysis using machine learning methods. In *Security in Computer Networks and Distributed Systems (SNDS)*, pages 440–450. Springer.

[Oktavianto and Muhardianto, 2013] Oktavianto, D. and Muhardianto, I. (2013). *Cuckoo Malware Analysis*. Packt Publishing Ltd., Birmingham.

[Paloalto Networks, 2016] Paloalto Networks (2016). What is malware? Retrieved from: https://www.paloaltonetworks.com/cyberpedia/what-is-malware.

[Pandey and Mehtre, 2014] Pandey, S. K. and Mehtre, B. M. (2014). A lifecycle based approach for malware analysis. In *4th International Conference on Communication Systems and Network Technologies*, pages 767–771.

[Park et al., 2010] Park, Y., Reeves, D., Mulukutla, V., and Sundaravel, B. (2010). Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, pages 45–49. ACM.

[Pascanu et al., 2015] Pascanu, R., Stokes, J. W., Sanossian, H., Marinescu, M., and Thomas, A. (2015). Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Petri, 2015] Petri, C. (2015). Decision trees. Retrieved from: http://www.cs.ubbcluj.ro/~gabis/DocDiplome/DT/DecisionTrees.pdf.

[Phatak, 2018] Phatak, M. (2018). Undersampling to achieve better recall. Retrieved from: https://www.kaggle.com/madhukaraphatak/under-sampling-to-achieve-better-recall.

[Pirscoveanu et al., 2015] Pirscoveanu, R. S., Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen, J. M., and Czech, A. (2015). Analysis of malware behavior: Type classification using machine learning. In *International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pages 1–7.

[Quist et al., 2011] Quist, D., Liebrock, L., and Neil, J. (2011). Improving antivirus accuracy with hypervisor assisted analysis. *Journal in computer virology*, 7(2):121–131.

[Radare2, 2018] Radare2 (2018). Radare2. Retrieved from: https://www.gitbook.com/download/pdf/book/radare/radare2book.

[Radware, 2017] Radware (2017). The history of malware. Retrieved from: https://www.radware.com/resources/malware_timeline.aspx.

[Raschka, 2014] Raschka, S. (2014). About feature scaling and normalization. Retrieved from: http://sebastianraschka.com/Articles/2014_about_feature_scaling.html#about-min-max-scaling.

[ReversingLabs, 2018] ReversingLabs (2018). Malware analysis platform. Retrieved from: https://www.reversinglabs.com/products/malware-analysis-appliance.html.

[Rhode et al., 2018] Rhode, M., Burnap, P., and Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. *Computers and Security*, 77:578 – 594.

[Rieck et al., 2011] Rieck, K., Trinius, P., Willems, C., and Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668.

[Robiah et al., 2009] Robiah, Y., Rahayu, S. S., Zaki, M. M., Shahrin, S., Faizal, M., and Marliza, R. (2009). A new generic taxonomy on hybrid malware detection technique. *arXiv preprint arXiv:0909.4860*.

[Santos et al., 2013] Santos, I., Devesa, J., Brezo, F., Nieves, J., and Bringas, P. G. (2013). Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS12-ICEUTE 12-SOCO 12 Special Sessions*, pages 271–280. Springer.

[Sarkar, 2018] Sarkar, D. (2018). Understanding feature engineering (part 1)-continuous numeric data. Retrieved from: https://towardsdatascience.com/understanding-feature-engineering-part-1-continuous-numeric-data-da4e47099a7b.

[Saxe and Berlin, 2015] Saxe, J. and Berlin, K. (2015). Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE.

[Saxe et al., 2013] Saxe, J., Mentis, D., and Greamo, C. (2013). Mining web technical discussions to identify malware capabilities. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 1–5. IEEE.

[Saxe et al., 2014] Saxe, J., Turner, R., and Blokhin, K. (2014). Crowdsource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, pages 68–75. IEEE.

[Schmidt et al., 2009a] Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K. A., Camtepe, S. A., and Albayrak, S. (2009a). Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE.

[Schmidt et al., 2009b] Schmidt, A.-D., Clausen, J. H., Camtepe, A., and Albayrak, S. (2009b). Detecting symbian os malware through static function call analysis. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 15–22. IEEE.

[Schultz et al., 2001] Schultz, M. G., Eskin, E., Zadok, F., and Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE.

[Searles et al., 2017] Searles, R., Xu, L., Killian, W., Vanderbruggen, T., Forren, T., Howe, J., Pearson, Z., Shannon, C., Simmons, J., and Cavazos, J. (2017). Parallelization of machine learning applied to call graphs of binaries for malware detection. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 69–77.

[Shafiq et al., 2009] Shafiq, M. Z., Tabish, S., and Farooq, M. (2009). Pe-probe: leveraging packer detection and structural information to detect malicious portable executables. In *Proceedings of the Virus Bulletin Conference (VB)*, pages 29–33.

[Shanks, 2014] Shanks, W. (2014). Enhancing incident response through forensic, memory analysis and malware sandboxing techniques. *SANS Institute, Mar*, 25(1):1–36.

[Shevchenko, 2007] Shevchenko, A. (2007). The evolution of technologies used to detect malicious code. Retrieved from: https://securelist.com/the-evolution-of-technologies-used-to-detect-malicious-code/36177/.

[Siddiqui et al., 2009] Siddiqui, M., Wang, M. C., and Lee, J. (2009). Detecting internet worms using data mining techniques. *Journal of Systemics, Cybernetics and Informatics*, 6(6):48–53.

[Storlie et al., 2014] Storlie, C., Anderson, B., Vander Wiel, S., Quist, D., Hash, C., and Brown, N. (2014). Stochastic identification of malware with dynamic traces. *The Annals of Applied Statistics*, 8(1):1–18.

[Stuart, 2005] Stuart, T. (2005). Instruction scheduling. Retrieved from: http://www.cl.cam.ac.uk/teaching/2005/OptComp/slides/lecture14.pdf.

[Swain, 2009] Swain, B. (2009). What are malware, viruses, spyware, and cookies, and what differentiates them? Retrieved from: https://www.symantec.com/connect/articles/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them.

[Symantec Corporation, 2017a] Symantec Corporation (2017a). What is malware and how can we prevent it? Retrieved from: https://us.norton.com/internetsecurity-malware.html.

[Symantec Corporation, 2017b] Symantec Corporation (2017b). When were computer viruses first written, and what were their original purposes? Retrieved from: https://us.norton.com/internetsecurity-malware-when-were-computer-viruses-first-written-and-what-were-html.

[Tang et al., 2011] Tang, Y., Xiao, B., and Lu, X. (2011). Signature tree generation for polymorphic worms. *IEEE Transactions on Computers*, 60(4):565–579.

[Thakur, 2016] Thakur, D. (2016). What is system call. Retrieved from: http://ecomputernotes.com/fundamental/disk-operating-system/what-is-system-call.

[ThreatExpert, 2017] ThreatExpert (2017). Retrieved from: http://www.threatexpert.com/.

[Tian et al., 2008] Tian, R., Batten, L. M., and Versteeg, S. (2008). Function length as a tool for malware classification. In *3rd International Conference on Malicious and Unwanted Software*, pages 69–76. IEEE.

[Tian et al., 2010] Tian, R., Islam, R., Batten, L., and Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. In *5th International Conference on Malicious and Unwanted Software*, pages 23–30.

[Tobiyama et al., 2016] Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., and Yagi, T. (2016). Malware detection with deep neural network using process behavior. In *40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582.

[Ugarte-Pedrero et al., 2015] Ugarte-Pedrero, X., Balzarotti, D., Santos, I., and Bringas, P. G. (2015). Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers. In *IEEE Symposium on Security and Privacy (SP)*, pages 659–673. IEEE.

[Vadrevu and Perdisci, 2016] Vadrevu, P. and Perdisci, R. (2016). Maxs: Scaling malware execution with sequential multi-hypothesis testing. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 771–782. ACM.

[Vanderbruggen and Cavazos, 2017] Vanderbruggen, T. and Cavazos, J. (2017). Large-scale exploration of feature sets and deep learning models to classify malicious applications. In *Resilience Week (RWS), 2017*, pages 37–43. IEEE.

[Wicherski, 2009] Wicherski, G. (2009). pehash: a novel approach to fast malware clustering. In *Proceedings of the 2nd USENIX conference on Large-scale Exploits and Emergent Threats*, pages 1–8.

[Willems et al., 2007] Willems, C., Holz, T., and Freiling, F. (2007). Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, pages 32–39.

[Yan et al., 2008] Yan, W., Zhang, Z., and Ansari, N. (2008). Revealing packed malware. *IEEE Security & Privacy*, pages 65–69.

[Yavvari et al., 2012] Yavvari, C., Tokhtabayev, A., Rangwala, H., and Stavrou, A. (2012). Malware characterization using behavioral components. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 226–239. Springer.

[Ye et al., 2007] Ye, Y., Wang, D., Li, T., and Ye, D. (2007). Imds: Intelligent malware detection system. In *Proceedings of the 13th ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1043–1047. ACM.

[Yin et al., 2007] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127. ACM.

[Zelenski and Parlante, 2008] Zelenski, J. and Parlante, N. (2008). Computer architecture: Take i. Retrieved from: https://see.stanford.edu/materials/icsppcs107/12-Computer-Architecture.pdf.

[Zeltser, 2001] Zeltser, L. (2001). How antivirus software works: Virus detection techniques. Retrieved from: http://searchsecurity.techtarget.com/tip/How-antivirus-software-works-Virus-detection-techniques.

[Zeltser, 2011] Zeltser, L. (2011). Process monitor filters for malware analysis and forensics. Retrieved from: https://zeltser.com/process-monitor-filters-for-malware-analysis/.