# ABSTRACT

Title of dissertation: Improving Existing Static and Dynamic Malware Detection Techniques with Instruction-level Behavior

Danny Kim
Doctor of Philosophy, 2019

Dissertation directed by: Professor Rajeev Barua
Department of Electrical
and Computer Engineering

My Ph.D. focuses on detecting malware by leveraging the information obtained at an instruction-level. Instruction-level information is obtained by looking at the instructions or disassembly that make up an executable. My initial work focused on using a dynamic binary instrumentation (DBI) tool. A DBI tool enables the study of instruction-level behavior while the malware is executing, which I show proves to be valuable in detecting malware. To expand on my work with dynamic instruction-level information, I integrated it with machine learning to increase the scalability and robustness of my detection tool. To further increase the scalability of the dynamic detection of malware, I created a two stage static-dynamic malware detection scheme aimed at achieving the accuracy of a fully-dynamic detection scheme without the high computational resources and time required. Lastly, I show the improvement of static analysis-based detection of malware by automatically generated machine learning features based on opcode sequences with the help of convolutional neural networks.

The first part of my research focused on obfuscated malware. Obfuscation is the process in which malware tries to hide itself from static analysis and trick disassemblers. I found that by using a DBI tool, I was able to not only detect obfuscation, but detect the differences in how it occurred in malware versus goodware. Through dynamic program-level analysis, I was able to detect specific obfuscations and use the varying methods in which it was used by programs to differentiate malware and goodware. I found that by using the mere presence of obfuscation as a method of detecting malware, I was able to detect previously undetected malware.

I then focused on using my knowledge of dynamic program-level features to build a highly accurate machine learning-based malware detection tool. Machine learning is useful in malware detection because it can process a large amount of data to determine meaningful relationships to distinguish malware from benign programs. Through the integration of machine learning, I was able to expand my obfuscation detection schemes to address a broader class of malware, which ultimately led to a malware detection tool that can detect 98.45% of malware with a 1% false positive rate.

Understanding the pitfalls of dynamic analysis of malware, I focused on creating a more efficient method of detecting malware. Malware detection comes in three methods: static analysis, dynamic analysis, and hybrids. Static analysis is fast and effective for detecting previously seen malware where as dynamic analysis can be more accurate and robust against zero-day or polymorphic malware, but at the cost of a high computational load. Most modern defenses today use a hybrid approach, which uses both static and dynamic analysis, but are subopti-

mal. I created a two-phase malware detection tool that approaches the accuracy of the dynamic-only system with only a small fraction of its computational cost, while maintaining a real-time malware detection timeliness similar to a static-only system, thus achieving the best of both approaches.

Lastly, my Ph.D. focused on reducing the need for manual feature generation by utilizing Convolutional Neural Networks (CNNs) to automatically generate feature vectors from raw input data. My work shows that using a raw sequence of opcode sequences from static disassembly with a CNN model can automatically produce feature vectors that are useful for detecting malware. Because this process is automated, it presents as a scalable method of consistently producing useful features without human intervention or labor that can be used to detect malware.

# Improving Existing Static and Dynamic Malware Detection Techniques with Instruction-level Behavior

by

Danny Kim

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Tudor Dumitras
Professor Charalampos (Babis) Papamanthou
Professor Michael Hicks
Professor Dana Dachman-Soled

# Dedication

I dedicate this work to Jesus Christ and my wife, Rebekah Kim. God first and foremost has always been foundational in my life and will continue to be the source of my motivation. My wife has been supporting me through the seeemingly endless years of getting underpaid, overworked, and denied from too many conferences. It was a grueling process filled with many doubts and failures, but she never lost faith in me. She has supported me with the unconditional love I needed to make it through. Though she did not contribute any research-related ideas, she provided laughs, love, and a balance in my life that I needed. She is the best wife and the perfect mate. This work and everything that comes of it is for you.

# Acknowledgments

I would first like to thank my advisor, Professor Rajeev Barua, for his unwavering guidance and timely advice in helping me complete this work. He taught valuable lessons in being detail-oriented, goal-minded, and results driven. I have learned an incredible amount from him as a teacher and a researcher that made my Ph.D. what it is.

Second, I would like to thank Professor Ankur Srivastava for pushing me to pursue my Ph.D. I would not be the same student or engineer without his relentless passion or guidance.

Third, I would like to thank Daniel Mirsky for his valuable help in the completion of my work. He was my constant sounding board for new ideas and thoughts. His ability to transform ideas into concrete code was unmatched and I could not have completed my work as quickly as I did without him.

Fourth, I would like to thank my friends through this process: Kevin Merchant, Michael Zuzak, Shang Li, and Amir Majlesi Kupaei. They have been with me through the entire process and were vital in keeping me sane and on track. I only hope that I am as helpful to them as they were to me.

Fifth, I would like to thank the Laboratory of Telecommunication Sciences and the ARCS foundation for funding parts of my research.

Lastly, I would like to thank Nikhil Kandpa, Vamsi Kovuru, Matthew Gilboy, Aaron Kramer, and Julien Roy for their individual contributions to my work. All of you played a part in the creation and development of the projects that made up

my Ph.D.

Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| AG | Alert Generation |
| API | Application Programming Interface |
| AR | Acrobat Reader |
| AUC | Area Under the Curve |
| AV | Anti-Virus |
| BBR | Blocked Benign Rate |
| BMR | Blocked Malicious Rate |
| CISC | Complex Instruction Set Computing |
| CNN | Convolutional Neural Network |
| CR | Computational Resources |
| CTI | Control Transfer Instruction |
| CV | Cross Validation |
| DEP | Data Execution Prevention |
| DBI | Dynamic Binary Instrumentation |
| DLL | Dynamically Linked Library |
| DPL | Dynamic Program-level |
| DR | Detection Rate |
| ESN | Echo State Network |
| FN | False Negative |
| FPR | False Positive Rate |
| IAT | Import Address Table |
| IDS | Intrusion Detection System |
| IOC | Indicator of Compromise |
| JVM | Java Virtual Machine |
| JIT | Just In Time |
| MLP | Multi-layer Perceptron |
| NLP | Natural Language Processing |
| NIST | National Institute of Science and Technology |
| NSRL | National Software Resource Library |

| | |
|---|---|
| OS | Operating System |
| PCA | Principle Component Analysis |
| PyInt | Python Interpreter |
| ROC | Receiver Operating Characteristic |
| RNN | Recurrent Neural Network |
| SOC | Security Operation Center |
| TIB | Thread Information Block |
| VM | Virtual Machine |

# Chapter 1: Introduction

Malicious software, better known as malware, is a problem in today's growing cyber community. Malware has continued to grow at an increasingly rapid rate, which brings the need for advanced and innovative defenses to an all-time high. Symantec reported discovering more than 430 million unique pieces of malware in 2015 alone [1]. The sheer number of malware with their growing complexities make detecting them a difficult task. Malware has been a cornerstone for cyberthieves and attackers to steal money, compromise information and undermine the security of all people.

Malware is decidedly a hard problem to solve. In order to combat advanced malware today, security companies use a combination of static and dynamic techniques for extracting unique indicators of maliciousness (IOM) from malware. Static techniques refer to analyzing a program without execution whereas dynamic techniques involve executing the program. These techniques are used typically in complementary fashions to extract malicious indicators that can be employed to differentiate malware from benign programs.

However, both current static and dynamic analysis methods fail to defeat advanced, obfuscated malware. Obfuscation is a technique malware writers use to

thwart the extraction of IOM during analysis and evade static detection. Examples of obfuscation are self-modifying code, and dynamically generated code. Obfuscation is effective against both static and dynamic analysis because it can hide the true control flow paths of a program. A control flow path of a program is the sequence of instructions or actions executed by the program when it executes. Often times, obfuscation is performed at an instruction level and thus can be difficult to detect or reverse engineer by using static analysis methods or OS-level dynamic analysis methods alone.

In order to detect and defeat obfuscation, a more exact tool is needed. In the first part of my work, I used a dynamic binary instrumentation (DBI) tool to analyze a program's instructions in real-time during its execution. With an instruction-level view of the program, obfuscations can be detected and analyzed. Understanding that malware employs different methods of obfuscation as a method of defeating both static and OS-level dynamic analysis, I used the presence of obfuscation as a method of detecting malware.

During my work, however, I found that obfuscation is not unique to malware. Benign programs (goodware) also used obfuscations for a variety of reasons. For example, I found that self-modification of instruction operands occurs in goodware because goodware populates jump tables dynamically at run-time. Thus, simply using the presence of obfuscation in a program as a method of detecting malware generated a high number of false positives. Related work that studied dynamic obfuscation detection in the past made no attempts to distinguish between malicious and benign obfuscations. To improve upon existing work, I had to create a set

of discriminating features that distinguished malicious obfuscations from benign ones. From literature, I implemented six different types of obfuscations, three with discriminating features, to prove that using the presence of malicious obfuscation is a viable method of detecting malware.

The first part of my work, detailed in Section 2, focuses on detecting obfuscations only in malware and describes in detail each detection technique. The work is culminated in a tool called DynODet. The section details DynODet's results when tested on a set of over 100K malware and over 6K goodware. It proves two key points. First, it shows that using the discriminating features for detecting malicious obfuscations is invaluable in maintaining a low false positive rate, making the tool deployable. Second, it shows that by analyzing these program-level behaviors, it was able to detect 25% of the malware that five prominent antivirus tools missed. This shows that particularly hard to detect malware may be employing advanced program-level obfuscations, which DynODet is capable of detecting.

Although DynODet showed promise in detecting previously undetected malware with obfuscation, it had two limitations. First, the discriminating features used to distinguish malicious obfuscations from benign ones was manually determined by analyzing the dynamic results of both malware and goodware. This is not a scalable method of discovering which obfuscations are specific to malware. Furthermore, manual analysis inhibits the usefulness of program-level behavior such as the frequency of specific opcodes because features such as these are less intuitive for humans to understand why they would occur more in malware versus goodware. Second, only 33% of the malware tested previously had one or more of the mali-

cious types of obfuscations studied. DynODet alone is not good enough to generally detect all types of malware.

To improve upon DynODet, I did two things. First, I integrated machine learning to produce a more scalable and accurate tool. This allowed me to enhance and expand my dynamic program-level feature set to include more obfuscations, more targeted behaviors, and general program-level behavior. Because I was no longer using ad-hoc discriminating features to identify obfuscations specific to malware, I was able to expand my dynamic program-level feature set to include more behaviors, which led to more detections.

This work specifically targeted two deficiencies I saw in modern dynamic analysis-based detection schemes. First, dynamic analysis is usually only run once per sample, which means dynamic analysis is based on the behaviors of a single execution path, which is often an underapproximation of a sample's total possible behaviors. This problem is exacerbated because current dynamic technologies that detect malware only focus on detecting how a program interacts with the OS as a method of monitoring behavior. This means they detect OS-level behavior, such as system calls or Windows library calls made by a program during its execution, and derive their IOCs based on the sequence of OS calls seen at run-time. I call these *external behaviors*. These external behaviors dismiss internal behavior, behavior that occurs in the program without the interacting with the OS, which I have shown is valuable in enhancing a tool's ability to reliably and robustly detect malware. Additionally, existing tools solely rely on OS calls or sequences of OS calls to determine malicious behavior. Existing dynamic analysis is adversely affected when

these specific malicious sequences of OS calls are not present in the single instance of behavior analysis. Second, dynamic analysis has been heralded to defeat obfuscation and packing because executing the program can reveal the OS calls made by the program. However, current tools cannot actually detect the methods of obfuscation or unpacking. They simply bypass the malware's defenses to try to uncover the OS calls made by the program. They miss the methods of how the malware hid their code and thus sacrifice a potential way of further differentiating malware from goodware.

I formally introduce Dynamic Program-level (DPL) information in Section 3. It covers how I use a DBI tool to detect more DPL information that proves to be both valuable and orthogonal to existing dynamic analysis information. To showcase the work, I built a malware detection tool that utilizes machine learning with a comprehensive feature set including my novel DPL features. To prove that DPL features are capable of improving existing dynamic analysis technologies, I detected two classes of DPL information: *potentially evasive* and *general program execution*. I define DPL potentially evasive actions as any dynamic program-level behavior that a program, malicious or benign, uses to hide the true code that is executed. I also detect three classes of general DPL information, which I define to be features that, although collected at an instruction-level, can quantify a program's dynamic execution characteristics.

My work shows that DPL features complement dynamic OS-level features in the following ways. First, it is gathered from monitoring the instruction sequences executed within the program. The use of DPL analysis gives access to wealth of

information that has not been used in previous literature for the purpose of malware detection. I show and explain in my work how DPL information is orthogonal and distinct from OS-level features. Second, examining DPL information enhances dynamic analysis's ability to correctly detect malware when the monitoring of OS-level calls produces little to no behavioral signatures. DPL features can be more fundamental to the nature of the program and my work shows that the information obtained, when combined with machine learning, can provably increase the detection rate of a OS-based tool.

After focusing on the dynamic analysis and detection of malware, I saw that existing detection architectures were not optimal in terms of performance and efficiency. Most modern malware defense systems rely on hybrid approaches, which combine both static and dynamic analysis in order to defend against malware. However, I found from a literature review that the way in which static and dynamic analysis have been combined is suboptimal. Most hybrid approaches work one of two ways. The first is a system which performs both static and dynamic analysis on all incoming traffic to obtain static and dynamic information in order to maximize a system's accuracy. The problem with this method is that although both sets of static and dynamic information are obtained, the computational and timeliness costs are prohibitively high because all programs are dynamically analyzed. The other primary hybrid approach is using a static analysis-based tool to detect any incoming threats. The detected threats, based on static analysis alone, are sent to a Security Operations Center (SOC) to be analyzed further by other methods such as dynamic analysis. This method is problematic because it relies solely on

the capability of static analysis to detect threats. Any active system maintains an extremely low false positive rate (to remain usable in the real world) by sacrificing its detection rate. Thus by only relying on static analysis, the malware detection system is incapable of defending robustly against all types of threats.

In Section 4, I present a new hybrid malware detection configuration that runs at the network entry point to an enterprise that strategically leverages the strengths of both static and dynamic analysis while minimizing their weaknesses.

I propose a detection system that contains a two-step process. First, a tool which I call a *static-hybrid* tool analyzes all incoming programs statically and categorizes them into one of three buckets: very benign, very malicious, and needs further analysis. It is thus named to distinguish it from usual static tools, which categorize incoming programs into only two buckets: malicious or benign. The very benign bucket contains programs that the static-hybrid is highly confident are benign. This bucket contains near zero false negatives (*i.e.*, missed malware). The very malicious bucket contains programs that the static-hybrid tool is highly confident are malicious. This bucket contains near zero false positives (*i.e.*, benign programs falsely detected as malware). The needs further analysis bucket contains programs the static-hybrid tool is unable to make a strong determination on and thus is passed to the dynamic analysis tool, which is the second step of the process.

The programs located in each bucket provide unique improvements to existing IDSs. The programs in the benign bucket can directly bypass the dynamic analysis portion of my tool, reducing the computational resources needed for dynamic analysis. The programs in the malicious bucket can be blocked immediately, improving

timeliness and reducing the need to conduct damage control after a program is identified as malicious in a passive IDS. The programs in the needs further analysis bucket are the only programs passed to the dynamic analysis tool. These programs, as my results will show, have a lower chance of being categorized correctly by a static analysis tool compared to a dynamic analysis tool. Thus by utilizing dynamic analysis on this subset of programs only, my tool's overall accuracy is higher than a strictly static IDS. In addition, because only the programs located in the very malicious bucket are immediately blocked, my static-hybrid tool is able to operate at a much lower false positive rate than a comparably built static-only tool, as my results will show.

In the process of building a static analysis-based malware detection tool, I found a few areas that despite the advancements in static analysis-based machine learning malware detection techniques still needed work. The first is the effort required to generate features that are useful for machine learning. Feature generation is vital in producing a highly accurate machine learning malware detection tool. This in some ways limits machine learning's capabilities to detect malware and goodware. Manual feature generation is a time-consuming process that still requires domain expertise and analysis to build effective features. Additionally, new features may have to be continually developed to deal with new malware. The second issue relates to the prominence of using features such as N-grams or other strictly presence-absence features that can cause overfitting resulting in less generalizable results [2]. Malware detection has largely moved away from traditional hash-based signatures, but using features such as N-gram combinations of Windows APIs can

be seen as a similar variant to hash-based signatures. They are only more effective because of the computing power of machine learning. More advances have to be made in order to reduce the time and knowledge necessary to create quality features and help improve a malware detection tool's ability to generalize and detect new malware.

To address the issues outlined above, I propose the use of static program disassembly and Convolutional Neural Networks (CNNs) in order to automatically generate machine learning features that are effective at differentiating between goodware and malware. I apply Natural Language Processing (NLP) methods to the static disassembly of both malware and goodware with the hypothesis that the opcode sequences found in programs can be treated as sentences would be in natural language processing. With the use of CNNs, I show that my tool is able to automatically generate features that embed raw opcode sequences without any manual intervention, and improve the accuracy of a tool based on existing static analysis features.

My work with CNNs and opcode sequences is distinct from related work because it strategically balances human knowledge of a Windows PE32 executable, knowing that opcode sequences are fundamental to the function and behavior of a program, with the power of automation of CNNs. The approaches found in the limited previous work that has looked at CNNs for the purpose of malware detection were suboptimal in this regard. Work such as [3] chose to train their CNN model on the first 2 million bytes of the input program as the input sequence, not taking into regard any understanding of inherit program structure. Although they produced reasonable accuracies, their model took over 1 month to train whereas our model

took hours with similar accuracies. Other works such as [4] relied too heavily on domain expertise on malware to produce an input sequence that proved to be useful, but required heavy manual analysis and development. My approach balances both understanding of program structure while minimizing the need for domain expertise.

The rest of the thesis is organized as follows. Section 2 covers the first part of my work into obfuscation detection. Section 3 covers the integration of machine learning and the expansion of my obfuscation detection, which results in a general malware detection tool. Section 4 focuses on improving the deficiencies of modern malware defense architectures. Section 5 looks into my work related to automatic feature generation based on opcode sequences.

# Chapter 2: Detecting Dynamic Obfuscation in Malware

## 2.1 Introduction

Malware writers have used a technique called obfuscation [5] to thwart extraction of such indicators of maliciousness (IOM) from malware and evade static detection [6]. Examples of obfuscation techniques are self-modifying code, and dynamic code generation. Although dynamic analysis tools have detected most types of obfuscation, obfuscation successfully thwarts static techniques such as [7] when analyzing malware. Obfuscated malware is a heavily studied subset within the field of malware [8]. The tools mentioned here [9–17] are capable of detecting and, in some cases, reversing obfuscation in malware. However, such tools are largely forensic tools and cannot be used for automatically distinguishing malware from benign programs because their schemes would detect obfuscation in both.

In this section, I study obfuscations present in programs with a goal of automatically distinguishing malware and benign programs. First, I present my study of the presence of six different obfuscation types in 6,192 benign applications. This study is the first that looks at which obfuscations (or code patterns that appear indistinguishable from obfuscation) are present in benign programs. Next, obfuscation is classified in two ways – allowed obfuscations (present in benign applications) vs.

disallowed obfuscations (usually only in malware). Through the study, I find that three of the six obfuscations I analyze are regularly present in benign applications. For these three, I create a set of *discriminating features* that reduce false positives. The remaining three obfuscation types are found to be largely restricted to malware. With my set of six obfuscations with discriminating features, I produce a malware detection technique, which is able to detect malware with high confidence. This includes the ability to detect previously missed malware using the presence of disallowed obfuscations as an IOM.

I present DynODet, a dynamic obfuscation detection tool built as an Intel Pin DLL [18] that detects binary-level obfuscations, and uses discriminating features to filter out benign application behaviors. When configured with discriminators, DynODet is not a general obfuscation detection tool and is not meant to generically detect obfuscation in all programs. Rather, DynODet is the first tool of its kind that can detect advanced malware, and hence is meant to be used in addition to existing detection tools. DynODet is meant for use in a sandbox [19], such as those in widely used sandbox based malware detectors. Enhancing the sandbox with DynODet increases malware detection without significantly increasing false positives. I present the following contributions:

- A unique study into the prevalence of obfuscations in benign applications
- Methods, implemented in my tool DynODet, to classify obfuscation types into two types - allowed obfuscations that are prevalent in benign programs and disallowed obfuscations that are present in malware
- Results showing that 33% of malware in a set of 100,208 have at least one of

the six disallowed obfuscations DynODet detects

- Results showing a false positive rate of below 2.5% in a test of 6,192 real world benign applications
- Results showing a decrease of 24.5% in malware missed by five market-leading AV tools by using DynODet's disallowed obfuscations

The section is structured as follows: Section 2.2 discusses the theoretical idea behind detecting obfuscation in malware. Sections 2.3 through 2.8 give an in-depth explanation of each of the six obfuscation types, including related work. Section 2.9 discusses DynODet's capabilities. Section 2.10 discusses current findings.

## 2.2 Detecting Dynamic Obfuscation

Obfuscation is defined as the deliberate attempt of programs to mislead a static analysis tool. Obfuscation works by thwarting or misleading static disassembly that is used in static analysis tools to understand the instruction-level structure of the program. Obfuscation has become a widespread tool in malware given its ability to defeat static analysis [6]. DynODet leverages the strength of dynamic analysis in order to improve static analysis efforts against obfuscated programs.

Although static analysis has been shown to be ineffective against obfucsation, it is still useful in determining a program's expected path. Using *just-in-time recurisive-traversal disassembly* is advantageous because it allows DynODet to produce a limited expected path of the program prior to its execution. Then during execution, for some of the obfuscations, *DynODet can compare the expected path to*

*the actual path to detect if any obfuscation is present.* DynODet implements just-in-time disassembly at run-time, which is the process of performing disassembly recursively during execution. It uses this in order to disassemble portions of the program just before they are executed in groups of code called *frontiers*. A frontier is the set of instructions reachable from the current instruction using direct control-transfer instructions (CTIs) only[1]. Hence frontiers terminate at a set of indirect CTIs. Frontiers are disassembled when execution reaches the current instruction at its beginning. Because indirect branch targets cannot be always determined until just before the instruction executes, just-in-time disassembly stops at all indirect branches. When an indirect branch target at the end of a frontier becomes known because the execution reaches that point, DynODet then restarts recursive traversal disassembly at its target.

DynODet chose to detect the following six obfuscations because these were among the obfuscations studied in academic papers, as shown through the related work sections below, and were discovered through my program-analysis of malware. The six chosen obfuscations are not an exhaustive list, but do show the potential of using the presence of obfuscations as IOMs. Also, by detecting these obfuscations, DynODet limits the ability of malware to escape static analysis.

[1] A CTI is called direct where the CTI's target is a constant; otherwise it is called indirect.

## 2.3 Self-Modification

### 2.3.1 Definition

Self-modifying code is when a program overwrites existing instructions with new instructions at run time. Self-modification defeats static analysis by producing instructions at run time that were not present during static analysis.


Figure 2.1: Example of Self-modification

Malware can implement this behavior by first changing the write permission on a region of existing code. Then the malware can overwrite the existing instructions with new instructions as seen in the example in figure 2.1. The malware can then re-execute the same region with the new instructions in place.

### 2.3.2 Presence in Benign Applications

When detecting self-modification by comparing an initial snapshot of a program with a current one after some execution time, I found that self-modification occurs in benign applications, in part because of dynamic address relocation. Dynamic address relocation is the process of updating a program with the runtime addresses for functions that exist in other DLLs or binary images. Static linking does not cause self-modification. In order to perform this relocation, the operating

system loader overwrites instruction operands that are dependent on the addresses of other DLL functions because the addresses is unknown statically. By naively implementing self-modification detection, this dynamic address relocation behavior would be flagged. DynODet uses the comparison of two snapshots of the program's code from different points of execution as its overall detection scheme, but also incorporates two discriminating features to distinguish malware from benign programs.

### 2.3.3 Detection Scheme

First, DynODet detects self-modification by comparing only the opcodes of instructions rather than the instructions along with their operands. I found that the dynamic linking process described above only modifies operands, whereas malware may modify both the operands and opcodes. Detecting only opcode modification reduces the detection of self-modification in benign programs while still detecting it in malware. By not being able to change the opcodes, malware is limited in its ability to perform malicious actions.

Second, DynODet does not flag a dynamic optimization found in a small percentage of benign programs as malicious. The dynamic optimization allows programs to overwrite the first instruction of a function with a JMP instruction. At run-time, a program may decide that it can reduce the runtime of the program by simply jumping to some other location every time this particular function is called [20]. A program can also overwrite a JMP instruction in order to enable a function to execute based on a runtime decision. Thus, DynODet allows a sin-

gle instruction per frontier of code to be replaced by a JMP instruction or have a JMP instruction replaced without it being considered malicious. DynODet's goal is not to generally detect self-modification, but to only distinguish self-modification in malware vs. benign applications.

DynODet does not detect self-modification in any dynamically allocated memory outside of the program's main image because it found this behavior in both benign applications and malware without a clear discriminating feature. During the course of execution, a program in Windows can allocate memory with read/write/execute permissions, which allows a program to use the region as a code cache where it can write code, execute it, and repeat. Benign interpreter programs will do this as confirmed in section 2.10.2.3.

At first, it may seem like malware can simply mimic benign programs in order to evade detection. However, it has been shown in studies such as [21] that static analysis of opcode sequences can be used to detect malware. Thus, malware can either only modify its instructions' operands and be detected by their opcode sequences, or it can overwrite its instructions and be caught by DynODet.

### 2.3.4 Related Work

Previous work, as explained below, detecting self-modification has largely focused on detecting it in a forensic setting, rather than using it as an IOM. Previous schemes would not be a viable detection tool because their methods of detecting self-modification would also detect it in benign applications.

PolyUnpack [22] detects self-modification by performing static analysis on a program, then comparing the dynamically executed instructions to those in the static disassembly. If they do not match, then PolyUnpack outputs the code. Mmm-Bop [23] is a dynamic binary instrumentation scheme that uses a code cache to help unpack a program and determine the original entry point. It detects self-modification by checking the write target of every write instruction to determine if the program wrote to a code cached instruction. However, both of these tools do not use the detection of self-modification as a method of catching malware, since their goals were malware understanding, not malware detection.

The following works [24–26] also detect self-modifying code, but do not propose their techniques as a method of detecting malware. [24, 26] in particular did not build their tools with the intention to use it on malware. [25] is a forensic tool and not meant for live detection. Work from the University of Virginia [27] provides a way, such as using a dynamically generated check-sum, to protect binaries against self-modification to preserve software integrity, but do not use the presence of self-modification as an IOM.

## 2.4 Section Mislabel Obfuscation

### 2.4.1 Definition

Section mislabel obfuscation is the process of dynamically changing the permissions of a section within the binary to execute a non-code region. By marking some sections of its binary as a non-code section, a static analyzer may not analyze

it for instructions, thus missing potentially malicious code.

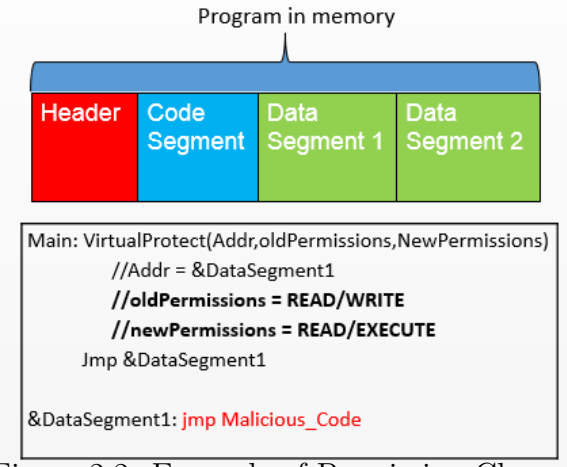

Figure 2.2: Example of Permission Change

As an example of section mislabel obfuscation, figure 2.2 shows a malware that first changes the permissions on a data section to executable, then it executes from the data region. This type of obfuscation allows malware to avoid statically marking sections of code in their binary, which can help evade static detection. It also allows the malware to possibly make changes to the non-code regions prior to changing the permissions on the section to further its obfuscation.

Self-modification, explained in section 2.3, can include the changing of permissions on a memory region. However, section mislabel obfuscation is distinct because it tracks when a malware intentionally mislabels a section or sections of a binary as a non-code region, only to later modify the permissions to execute the section. Self-modification, per my definition, only occurs in code sections.

### 2.4.2 Presence in Benign Applications

From my study of benign applications, section mislabel obfuscation does not occur in most benign programs. This is most likely due to the use of standard compilers when compiling benign applications. Thus, DynODet does not employ a discriminating feature for this obfuscation.

### 2.4.3 Detection scheme

DynODet detects section mislabel obfuscation by using Pin's API to determine a program's sections when first loaded into memory. Each binary has several sections such as code, and data. DynODet stores which address ranges are marked as code regions and which are not. It then monitors the execution of the program and if the PC lies within a non-code region then section mislabel obfuscation has occurred. DynODet does not detect the actual request of the program to change the permissions on its section, but is still able to determine if such an event did occur by watching the execution of the program.

Because DynODet does not employ a specific discriminating feature in order to detect this obfuscation, there is no way for malware to hide this obfuscation.

### 2.4.4 Related Work

I am not aware of any tool that explicitly detects section mislabel obfuscation in the manner that DynODet does, but there are existing schemes that try to prevent the execution of non-code sections.

Windows and other OSs have implemented a protection called data execution prevention (DEP) [28]. DEP prevents programs from executing code from non-executable memory regions such as data sections. Although it seems DEP employs a similar goal to my method, the goals are not identical – DEP is primarily meant to prevent hijacking of control of critical Windows system files using data execution, for example in a stack smashing attack. DynODet aims to detect malware payload files. Consequent to its goals, most DEP implementations do not prevent adding execute permissions to segments. DynODet will detect such permission changes. Another drawback of DEP with regard to DynODet's goals is that with DEP, if a piece of malware on an end point performs some malicious actions prior to attempting to execute data, then those prior will be allowed. In contrast, DynODet is meant to be an integral part of a sandbox mechanism, which means detection of section mislabeling will imply that the malware will be prevented from reaching the end point in its entirety.

## 2.5 Dynamically Generated Code

### 2.5.1 Definition

Dynamically generated code is the process of creating code in a dynamically allocated memory region. Malware dynamically generates code because it wants to hides its malicious instructions from static analysis. As in figure 2.3, malware can first allocate a new region of memory with read/write/execute permissions. It can then copy over instructions to the new memory region and then execute

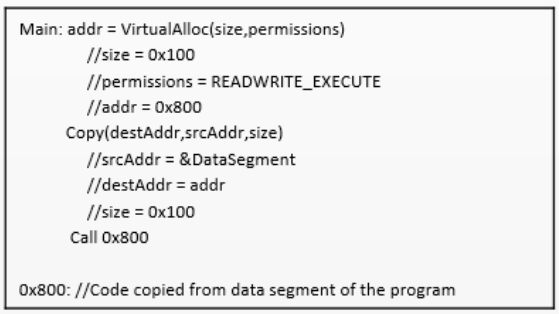

Figure 2.3: Example of Dynamically Generated Code

it. To add a level of obfuscation, malware could also decrypt a data section that holds malicious instructions prior to copying over the instructions. Static analysis is defeated here because it cannot reliably decrypt the data section to reveal the instructions in addition to its inability to know with high confidence that the data section is encrypted.

Dynamically generated code differs from self-modification because dynamically generated code is the action of allocating new memory, copying code to it, then executing that region. Self-modification refers to overwriting existing instructions (*i.e.* instructions that have already executed) with new instructions and then executing the new instructions.

### 2.5.2 Presence in Benign Applications

In my experiments, I found that benign applications also dynamically generate code for a variety of reasons. For example, I found that some benign programs generate jump tables that can only be built after the linking of other DLLs at runtime because the addresses of DLL functions are not known statically. Benign applications may also copy a function to a new region of memory to create a wrapper

function. In the cases where a benign program dynamically generates code, I found that the code that is copied to new memory is in an existing code section of the binary image loaded at run time. This is because the dynamically generated code that is copied to new memory is often generated by a compiler, which naturally places code in a code section. Understanding that dynamically generated code occurs in benign applications, discriminating features are needed in order to eliminate false positives.

### 2.5.3 Detection Scheme

DynODet detects dynamically generated code in a three-step manner. First it hooks into Windows systems calls that are related to the allocation and changing of permissions of memory. DynODet begins to track any region that is marked as executable and is not a part of any loaded binary image. Second, it instruments any write instructions to tracked memory regions from the program so that right before it executes, DynODet can determine if such a memory region is written to. If such a write occurs, DynODet checks to see if the source address of the write instruction is from one of the program's non-code regions. If so, DynODet watches for the newly copied code to execute at which point DynODet detects dynamically generated code.

With DynODet's unique method of detecting dynamically generated code, malware cannot simply try to mimic the behavior of benign applications in order to evade detection. If the malware tried to specifically evade DynODet's detection, it

would only be allowed to copy code into the newly allocated region of memory from statically-declared code sections. If the code is copied from a code section, then the code is discoverable statically and thus defeats the purpose of dynamic code generation. In regards to self-modification in external memory regions, if the code that is replacing existing code is from a data section inside of the main image of the binary, then DynODet will detect it. If the code is from a code section, then static analysis can discover it.

Just-in-time (JIT) compilation and interpretation of code are two types of programs that are closely tied to dynamically generated code. JIT compilation and interpretation of code are present in platforms that take an input of data or bytecode and translate it into machine code. Unfortunately, interpretation of code is a powerful tool that malware can misuse. For example, malware can use tools such as Themida and VMProtect to obfuscate attacks [16]. Further research is needed to mitigate this risk. The current implementation of my detection scheme for dynamically generated code with discriminating features does not flag interpretation or JIT compilation as malicious. However, one potential solution to this problem is to whitelist benign interpreter and JIT-platform programs, which is feasible given their small number and well-known nature- this has the added benefit of preventing the malicious use of interepreters unknown to the user.

### 2.5.4 Related Work

As noted above, detecting and tracking dynamically generated code is a solved problem. However, none of the following tools are able to use their detection of dynamically generated code to catch malware.

OllyBonE [15], a plug-in to OllyDbg [29] is a kernel driver which reports when pages are written to then executed. However, OllyBonE was only tested on common packer code and not on benign applications. OmniUnpack [30] is a similar tool that tracks memory page writes and then analyzes the contents when a dangerous system call is made. When such a call is made another malware analysis tool is invoked to analyze the written pages. ReconBin [31] also analyzes dynamically generated code. Renovo [32] attempts to extract hidden code from packed executables by watching all memory writes, and determining if any instructions executed are generated. It does not check the source of the writes and was not tested on benign applications. None of these tools use the presence of dynamically generated code as an IOM, since, lacking my discriminant, they would have found such code in several benign applications as well. These tools are also limited in their malware analysis only to dynamically generated code.

## 2.6 Unconditional To Conditional Branch Obfuscation

### 2.6.1 Definition

Unconditional to conditional branch obfuscation is the process of converting an unconditional branch to a conditional branch by the use of an opaque predicate [33]. An opaque predicate is an expression that always evaluates to true or false and thus can be used in a conditional branch to always transfer control one way. This obfuscation can be used to thwart static analysis by adding more control-paths that static analysis has to analyze.

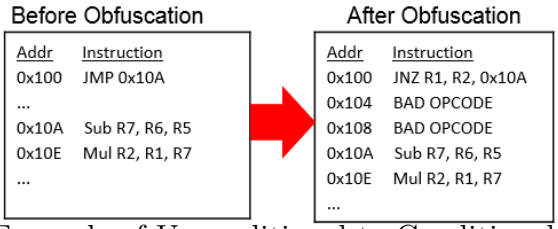


Figure 2.4: Example of Unconditional to Conditional Obfuscation

For example, in figure 2.4, malware can take an unconditional branch and convert into a conditional branch. The malware can set the R1 and R2 registers such that they are never equal thus always jumping to the target address of `0x10A`. Now the malware can insert junk code or invalid opcodes at the always not-taken direction of the conditional branch in order to confuse the static disassembler.

This type of obfuscation is generally defeated by any dynamic analysis tool by watching the actual execution of the program, and seeing which paths were taken and never taken. *However, merely having one outcome of a conditional branch never execute is not indicative of the presence of this obfuscation, since benign program*

*may have branches outcomes which never happen (for example error checks that never trigger in a particular run).* In order to detect this obfuscation specifically, some additional analysis has to be done.

### 2.6.2 Presence in Benign Applications

Although benign programs rarely use this obfuscation, having one side of a branch in a benign program never execute is common. Thus, I need a discriminating feature to distinguish this behavior in malware vs. benign applications.

### 2.6.3 Detection Scheme

To distinguish malware from benign applications, DynODet uses the observation that if malware uses this obfuscation, then the untaken path is likely not code, and can be detected as such by just-in-time disassembly. DynODet disassembles both the target and fall through address of each conditional branch until an indirect branch is reached. If either control flow path contains an invalid instruction prior to reaching an indirect branch, this obfuscation is detected. DynODet stops inspecting the control flow paths at indirect branches because the targets of indirect branches are unknown statically.

This is as far as DynODet is able to detect unconditional to conditional branch obfuscation because it is hard to determine whether code is doing useful work or not. However, it does eliminate the malware's ability to place junk code at either the target or fallthrough address of a conditional branch, which can thwart some

static disassemblers.

### 2.6.4 Related Work

DynODet is not aware of any work that explicitly detects this obfuscation. In most dynamic analysis tools, this obfuscation is partially detected as a by-product, but is not documented as being used to distinguish malware from benign programs.

## 2.7 Exception-Based Obfuscation

### 2.7.1 Definition

Exception-based obfuscation is the process of registering an exception handler with the OS then causing an exception intentionally. Static analyzers fail to see this because exceptions can occur in programs in ways that are not statically deducible. For example, a divide instruction, which usually take registers as operands, can have the divisor equal to zero. It is very difficult for static analysis tools to reliably confirm that the divisor register will be set to zero.

```
Main: //register Div_trap_handler function as exception handler
      call Foo

Foo:  Add  R1, R2, R3
      Div  R4, R5, R6     /* R4 = R5 ÷ R6 */
      Xor  R7, R8, R9
      ...

Div_trap_handler: Jmp Harmful_function
```

Figure 2.5: Example of Exception-based obfuscation

An example is shown in figure 2.5. The malware first registers an exception handler that jumps to harmful code. Then, the malware causes an intentional

exception such as division by zero. In figure 2.5, R6 would be equal to zero. The registered exception handler will then execute and the program will jump to the harmful code. This instruction execution sequence would not have been predicted or analyzed by static analyzers due to the dynamic nature of exceptions.

### 2.7.2 Presence in Benign Applications

Exception-based obfuscation has been studied in the past as the related work section below shows, but I am not aware of any work that uses this as an IOM for malware. Unsurprisingly, handled exceptions do occur inside of benign applications. Legitimate exception handlers may execute because of an invalid input from the user or bad input from a file. Because of this, simply classifying any exception handler execution as exception-based obfuscation is not a viable detection technique.

### 2.7.3 Detection Scheme

Benign applications do not intentionally cause exceptions, such as divide by zero, in their programs because these would cause a system error. Malware, however, to ensure that its malicious exception handler executes, will cause an intentional exception. Using this, DynODet incorporates the following discriminating features to catch exception-based obfuscation in malware.

DynODet detects exception-based obfuscation by monitoring exception handler registration followed by an exception occurring. During execution, DynODet monitors two methods of registering an exception handler. One is the standard

Windows API *SetUnhandledExceptionFilter*. The other method is directly writing to the thread information block (TIB). In order to detect this method, DynODet watches all writes to the TIB and detects any changes to the current exception handler. Once an exception handler is registered, DynODet instruments the beginning of it and is notified when it runs. Next, DynODet strengthens the probability of detecting a malicious exception by catching an *unexpected control flow exception*. DynODet defines an unexpected control flow exception as when the control-flow of the program does not follow the expected path. In order to detect an unexpected control flow exception, DynODet instruments every Pin basic block and keeps track of the first and last instruction addresses. Prior to the dynamic execution of each basic block, DynODet checks if the entire prior basic block executed. A basic block is a sequence of instructions that should execute from top to the bottom, unless an exception occurred. If DynODet determines that a previous basic block did not execute in its entirety, it turns an internal flag on. If the next basic block executed is in a registered exception handler and the internal flag is on, exception-based obfuscation has occurred. The internal flag is used because DynODet is only concerned with exceptions that occur within the application because malware, when employing this obfuscation, triggers the exception in its code to ensure that the exception will occur.

This type of obfuscation is rare, as will be seen in the results below. The point of this obfuscation is to hide a malware's true path of execution from a static analysis tool. If malware chose to implement exception-based obfuscation in a manner that deliberately evades DynODet's detection scheme, such as by using an explicit

interrupt instruction, the interrupt could be discovered through static analysis, thus making the obfuscation less useful. DynODet's detection scheme allows it to broadly detect most exception-based obfuscation scenarios such as divide by zero or writes to address zero.

### 2.7.4 Related Work

There have been a few works that looked at how exception-based obfuscation may be present in malware, but none have created a general solution and used it as an IOM. Prakash [34] introduced an x86 emulator that attempts to detect exception-based obfuscation in attempts to generically unpack malware. The emulator only detects common exception-based obfuscations, such as using the x86 interrupt instruction or a divide by zero, to ensure that the emulator continues running. It does not use exception-based obfuscation as a detection mechanism and did not study it in benign programs. Work from the University of Arizona [35] proposes using exception handlers as a method of obfuscation, but does not propose or deliver a mechanism of detecting it.

## 2.8 Overlapping Code Sequences

### 2.8.1 Definition

Overlapping code sequences occur when jumping into the middle of an existing instruction. This method allows malware to hide instructions within other instructions, which can trick static disassemblers. In the complex instruction set

computing (CISC) x86 architecture, instructions have variable length so the same range of addresses can have several different instruction sequences.

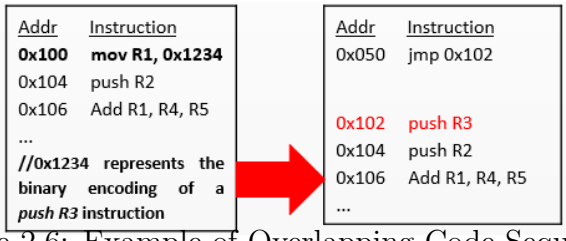

Figure 2.6: Example of Overlapping Code Sequences

As shown in figure 2.6, to implement this behavior, a malware can use an instruction with an immediate value that is an encoding of an instruction. A static disassembler will not be able to see the hidden instruction because it is not located at a natural instruction address and could miss malicious behavior. A push instruction is represented in the immediate value of a MOV instruction as a proof-of-concept in figure 2.6.

### 2.8.2 Presence in Benign Applications

During benign testing, DynODet found a very small number of benign programs with this behavior. To the best of my knowledge, there are no high-level construct that would produce this behavior when compiled by a standard compiler. Thus, no discriminating feature is needed to use this as an IOM.

### 2.8.3 Detection Scheme

During disassembly, DynODet checks if the current PC is in the middle of an existing instruction in the disassembly map produced by recursive traversal. If it

is, DynODet checks if all bytes of the current instruction match the original code. If they do not match, self-modification has occurred. If they do, overlapping code sequences are present.

### 2.8.4 Related Work

Work at the University of Louisiana at Lafayette [36] implements segmentation to produce a collection of potentially overlapping segments, where each segment defines a unique sequence of instructions. Their tool attempts to uncover all potential sequences of instructions. Their tool does not provide much insight into which sequence of instructions actually ran, and provides superfluous instructions that may never be executed dynamically. Their scheme also does not try to detect overlapping code sequences as an IOM.

## 2.9 DynODet Capabilities and Limitations

### 2.9.1 Capabilities

#### 2.9.1.1 Multithreaded Applications

: DynODet can handle multithreaded applications so long as the underlying dynamic binary instrumentation (DBI) tool it uses (such as Pin) handles multithreading. Pin assigns each created thread a thread ID that is used to distinguish which thread is executing and being instrumented. DynODet analyzes each thread's dynamic execution separately, and combines all detections found in all threads.

#### 2.9.1.2 Spawned Child Processes

: DynODet is able to handle programs that spawn other processes. This is a common behavior for many programs so DynODet is injected by Pin into each child process and each process gets its own unique analysis, but ultimately gets grouped into the parent process's analysis.

### 2.9.2 Limitations and Assumptions

Malware detection is a constant arms race between the malware writers and cybersecurity companies. As with any new scheme, malware can specifically target to defeat my scheme and bypass my detections. However, my detection schemes do make it considerably more difficult for malware to bypass both conventional static detection and my tool's detections. I address these limitations at the end of this section.

## 2.10 Results

### 2.10.1 Test Set up

My tool is currently built as an Intel Pin [18] dynamically linked library (DLL). Pin is a DBI tool that provides all of the necessary capabilities described above. Although the current implementation of DynODet is tied to Pin, my detection mechanisms are universal and can be implemented using other dynamic binary rewriters. As a Pin DLL, DynODet gets injected into the program being monitored in a sand-

box environment. After the program is finished running or killed by my timeout, the results are collected and sent back to the host computer.

The Cuckoo Sandbox [37] is a popular malware analysis environment that provides an infrastructure for dynamic malware testing. I perform all of my malware testing in duplicated Cuckoo sandboxes, one sandbox per malware, which are reverted to a clean environment prior to each execution. I give my Cuckoo Sandbox process 32 KVM [38] instances, which are running Windows 7 with 1GB of RAM each. I also set up INetSim [39] in order to give the KVM machines a fake Internet connection to trick malware into thinking it has access to Internet. INetSim is used here to get better code coverage in malware. DynODet has an internal timeout of one minute per malware.

### 2.10.2 Benign Applications

I performed benign application testing for 6,192 Windows programs. In order to ensure that my testing was comprehensive, I tested two sets of benign programs. First were 119 programs that had to be installed prior to executing (referred to as the installed benign set). Second was a set of 6,073 benign programs from a list produced by the National Institute of Standards and Technology (NIST) [40].

For the installed benign set, some of the installed programs tested were Adobe Acrobat, Mozilla Firefox, QuickTime Player, Notepad++, Google Chrome, WinScp, Open Office, 7zip, and Java Virtual Machine (JVM). The other installed programs were a mix of media players, music players, text editors, IDEs, mail clients, digital

recorders, and standard system tools. When testing these installed benign applications, there was no easy method of interaction automation because these programs require a complex level of input. A generic interaction script cannot be created to robustly test all of the benign applications. In my best attempt, I modified human.py, Cuckoo's python script that is responsible for simulating human interaction, to click in a grid pattern, left to right, top to bottom, in fixed intervals across the window of the benign application in order to click as many buttons as possible. Human.py also input random text in order to give input into text fields of applications. Although this was a simple method of interaction, the purpose of this was to increase code coverage in each application.

For the second part of my dataset, I obtained a list of Windows executables from the National Software Reference Library (NSRL) produced by NIST. The NSRL set contains millions of hashes of benign software. From the initial list, I extracted the unique hashes of Windows 7 compatible executables and queried Virus Total in order to download the binaries. This resulted in 6,073 benign applications that I was able to test. The NSRL Set is largely considered to be a set of known benign software as noted in [41, 42]. However, there is a small subset of known hacker tools and other unknown software that are considered to be malicious [43]. For testing purposes, I removed these programs to ensure that my benign dataset was truly benign in order to evaluate my tool properly. Ground truth is important, thus I felt that the preceding precautions were justified. Additionally, due to the sheer number of samples, it was not feasible to test and install each by hand. The ability to thoroughly test benign applications that arrive as standalone executables

is outside of the scope of DynODet.

#### 2.10.2.1 NSRL Set

The results for the 6,073 programs from the NSRL set are listed below in table 2.1. In table 2.1, the second column shows the obfuscation types that were detected in benign applications when no discriminating features were implemented. Without discriminating features, 8.35% of benign programs tested contain at least one type of obfuscation. The third column shows the obfuscations present with discriminating features. With discriminating features, the false positive rate is reduced by nearly 70% for programs with one or more obfuscations to 2.45% and 75% for programs with two or more obfuscations to .13%. As with any detection tool, the false positive rate is important and using these obfuscation types without discriminating features as IOM of malware is not viable.

The 149 programs falsely flagged, from manual inspection, seem to have no single attribute that explain their false detection. Using PeID, a popular packer identifier, I was able to determine that 58 of the 149 programs were packed with a variety of packers and compressors. My conjecture is that these programs are very uncommon and do not follow standard compilation tools as supported by my testing results. These programs, rather than intentionally implementing unallowed obfuscations, are most likely flagged due to some extreme corner cases that my current implementation does not allow.

The subset of programs tested from the NSRL set were obtained from Virus

Total. This leads me to believe that the programs tested in this dataset are more representative of the types of executables that would be tested in an intrusion detection system. Thus, the results here show the versatility of DynODet in that it can work for both large, installed programs as well as standalone executables such as those arriving at a network's firewall.

#### 2.10.2.2 Standard Installed Applications

The results for 117 out of the 119 applications are also listed in table 2.1. Only 117 were tested here because two of the programs, JVM and Python interpreter, cannot be run without input. These are tested and explained in the next sub-section. Out of the 117 benign applications tested, only one program had a false positive in any of my obfuscation detectors, namely a section mislabel obfuscation. The program with the false positive was a graphical viewer called *i_view.exe*. This false positive is caused by Pin's inability to detect a code region along with the program's possible use of a non-standard compiler that may have produced an irregular header. As shown in Table 2.1, none of the indicators outside of the one explained above, were detected in 117 benign applications. This shows that the modifications that were made in DynODet reduce false positives for three of the indicators to nearly 0%.

#### 2.10.2.3 Interpreters and JIT-platform Programs

I also studied interpreter programs, which are programs take in data containing executable code. For example, Adobe Acrobat Reader (AR) takes in a PDF as input, which can contain executable code. *DynODet's goal is not to detect malicious PDFs.* Rather, I aimed to find out whether DynODet detects behaviors such as dynamically generated code in the interpreter program.

Each interpreter program was tested with a small set of inputs. AR was tested with 12 PDFs that included scripts, such as a template form. The Python interpreter (PyInt) was tested with a set of nine python scripts that performed small tasks such as analyzing a directory for mp3s or printing the date and time. Firefox was tested with eight websites running javascript such as `www.cnn.com`, `www.youtube.com`, and `www.amazon.com`. JVM was tested with 11 benchmarks out of Decapo Benchmarks, a Java open-source benchmark suite [44].

As table 2.2 shows, when my tool did not use discriminating features, it detected dynamically generated code in AR, Firefox and JVM. With my discriminating features incorporated, there were no detections. This proves that my discriminating features are valuable in not detecting obfuscation in benign applications.

As noted in section 2.3.2, I did not test for self-modification in dynamically allocated memory. I found that Firefox and JVM allocate memory outside of their main images to use as a code cache when executing chunks of interpreted code. As mentioned in section 2.5.3, there are other methods of detecting malicious interpreter and JIT-platform programs.

### 2.10.3 Malware

The malware samples were collected by my group from Virus Total, a database of malware maintained by Google. I tested in total of 100,208 malware selected randomly from 2011 to 2016. The malware test set used is a comprehensive set including viruses, backdoors, trojans, adware, infostealers, ransomware, spyware, downloaders, and bots. It is worth noting that Virus Total does contain some benign applications as well; hence I only tested samples that had at least three detections in their corresponding Virus Total report to filter out false positives.

As seen in Table 2.3, DynODet found examples of each obfuscation in malware. The table shows the number of detections in the 100,208 malware tested when discriminating features are used. With discriminating features enabled, 32.7% of malware are detected. *DynODet here is not claiming that the results below show the true number of malware that have these characteristics.* Rather, DynODet is showing the number of malware that can be detected despite having made modifications to some of the dynamic obfuscation detection schemes.

Although some of the detections such as overlapping code sequences and exception-based obfuscation were not that common in malware, it is still useful to include them as malware detectors for two reasons. The first is that these are rarely found in benign programs so adding to the list of distinguishable characteristics between malware and benign applications will always be useful. Second, an advantage of DynODet is that it uses all of these detections in combination in order to catch malware. Thus, although the detections of individual obfuscations may be

small, when combined, they can be substantial.

As seen in Table 2.3, DynODet found that 32.74% of malware tested had at least one disallowed obfuscation. When compared to benign programs, in which less than 2.5% had at least one indicator, there is a clear distinction between malware and benign programs. This allows DynODet to be employed as a detection tool, rather than just an analysis tool. If a use case of DynODet could not tolerate any false positives, then it can be altered to only classify programs with 2 obfuscations as malware, which still results in a 5.74% detection rate.

Another indication of the capability and novelty of DynODet is shown in Table 2.4. I analyzed the detections of the following five market-leading AV tools: Kaspersky, ClamAV, McAfee, Symantec, and Microsoft Security Essentials and gathered the subset of malware that were not detected by any of the tools. The set resulted in 12,833 malware. I was able to show that DynODet is able to detect 4,445 (34.63%) without discriminating features and 3,141 (24.48%) with discriminating features out of the previously missed malware. This also shows the efficacy in using obfuscation detection in order to detect malware that was previously hard to catch. The detection rate of each tool listed below was obtained through Virus Total's reports for each malware.

### 2.10.4 Limitations with Evasion

As with any detection scheme, there are ways for malware to evade my tool specifically. I have listed below possible evasion techniques for three of the obfusca-

tions detected in DynODet.

- Section mislabel obfuscation: In order to evade section mislabel detection, malware can mark all sections in their program executable so that regardless of which section it executes from, it will not be caught by my detection. However, this becomes problematic for the malware for two reasons. First, from my analysis of benign programs, I found that almost no program had all executable sections. If malware, in order to evade my detection scheme, started to mark all sections executable, this would be an easy sign for analysis tools to pick up on. Second, if malware marks all sections as code, every analysis tool will analyze all of its sections expecting code. This leads to two scenarios. Either the malware's malicious code will be revealed, or in attempts to hide its code, the malware's code sections will have high entropy due to encryption or no valid code at all, which is suspicious and will also be caught by detection schemes.
- Exception-based obfuscation: My tool currently detects all hardware exceptions that lead to the execution of a registered exception handler as a potentially malicious indicator. Although hardware exceptions can occur in benign programs, through my study, I found that the frequency of occurrences in benign programs differ from those in malware. As supported by my data, this occurs about three times more often in malware than in benign applications. Although this is not a great indicator, I conjecture that it might be useful in a machine-learning framework as one feature among many used to weigh the likelihood of a program being malicious.

- Unconditional to conditional obfuscation: Malware can evade this detection by putting legitimate code at every conditional branch in the program. As mentioned in section 2.6.3, DynODet is unable to prevent this evasion. However, it does constrain the malware writer's flexibility in placing data in the code section, since not all data values also represent valid instructions. Moreover, it increases the work of the malware writer.

## 2.11 Conclusion and Future Work

DynODet has exemplified that there is a way to use dynamic program-level analysis a method of detecting and differentiating benign and malicious obfuscations. DynODet also showed that dynamic program-level analysis can be key in detecting previously unseen malware, as shown by the 25% detection rate of unknown malware.

However, DynODet has two major limitations. First, the discriminating features used were determined through manual analysis of benign programs and malware. This is not a scalable or robust method of differentiating obfuscations in malware and benign programs. Furthermore, using the presence of a single disallowed obfuscation as an indicator of malware is not realistic. Second, DynODet could only detect disallowed obfuscations in 33% of malware. Although DynODet is not a standalone tool, it still does not detect a large number of malware, making it less useful in the real world. To address these shortcomings, the next section of my proposal addresses how to integrate machine learning and expanding the feature set to build a highly accurate, robust detection tool.

| Detection | W/O discriminating features | | W/ discriminating features | |
|---|---|---|---|---|
| | NSRL | Installed | NSRL | Installed |
| Self-modification | 135 | 5 | 2 | 0 |
| Section Mislabel | 51 | 1 | 51 | 1 |
| Dynamically Generated Code | 296 | 29 | 86 | 0 |
| Unconditional to Conditional | 12 | 0 | 12 | 0 |
| Exception-based | 27 | 1 | 5 | 0 |
| Overlapping Code Sequences | 5 | 0 | 5 | 0 |
| **Had 1 or more obfuscations** | **507/6,073 (8.35%)** | **34/117 (29.05%)** | **149/6,073 (2.45%)** | **1/117 (.85%)** |
| **Had 2 or more obfuscations** | **33/6,073 (.54%)** | **2/117 (1.71%)** | **8/6,073 (.13%)** | **0/117 (0%)** |

Table 2.1: Benign Application Results

| Detection | W/O discriminating features | | | | W/ discriminating features | | | |
|---|---|---|---|---|---|---|---|---|
| | AR | PyInt | Firefox | JVM | AR | PyInt | Firefox | JVM |
| Self-modification | 0/12 | 0/9 | 0/8 | 0/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| Section Mislabel | 0/12 | 0/9 | 0/8 | 0/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| Dynamically Generated Code | 12/12 | 0/9 | 8/8 | 11/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| Unconditional to Conditional | 0/12 | 0/9 | 0/8 | 0/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| Exception-based | 0/12 | 0/9 | 0/8 | 0/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| Overlapping Code Sequences | 0/12 | 0/9 | 0/8 | 0/11 | 0/12 | 0/9 | 0/8 | 0/11 |
| **Had 1 or more obfuscations** | **12/12 (100%)** | **0/9 (0%)** | **8/8 (100%)** | **11/11 (100%)** | **0/12 (0%)** | **0/9 (0%)** | **0/8 (0%)** | **0/11 (0%)** |

Table 2.2: Benign Interpreter Results

| **Detection for 100,208 Malware** | | |
|---|---|---|
| Detection | W/ discriminating features | |
| Self-modification | 10,264 | 10.24% |
| Section Mislabel | 19,051 | 19.01% |
| Dynamically Generated Code | 7,106 | 7.09% |
| Unconditional to Conditional | 7,889 | 7.87% |
| Exception-based | 334 | 0.33% |
| Overlapping Code Sequences | 1,710 | 1.71% |
| **Had 1 or more obfuscations** | **32,811** | **32.74%** |
| **Had 2 or more obfuscations** | **5,750** | **5.74%** |

Table 2.3: Obfuscation in Malware Results

| **Detection of 12,833 Missed Malware** | | | |
|---|---|---|---|
| W/O discriminating features | | W/ discriminating features | |
| **4,445** | **34.64%** | **3,141** | **24.48%** |

Table 2.4: Malware Detection Improvement

# Chapter 3: Enhancing Dynamic Analysis-based Malware Detection with Dynamic Program-level Features

## 3.1 Introduction

As I look to build upon DynODet and expand my work, I looked at how dynamic analysis and detection is done today in order to improve upon the status quo.

There are two primary methods of malware detection. The first is sandbox-based detection, also known as dynamic analysis. Sandbox-based analysis aims to reveal malicious behavior by executing a program within a protected environment. Sandbox-based analysis is effective because some malware are packed or obfuscated in some way that make static analysis harder to reliably perform. Static analysis is the process of analyzing a program without execution and has been shown to be defeated by packing or obfuscation [6]. Dynamic analysis executes the program and can monitor its actual behavior, but is computationally intensive and slow (operates on the order of minutes). Both analysis types have their strengths and weaknesses and I make no claim as to which is better. Rather they are used for different purposes – static analysis is usually used for large-scale rapid detection of malware, whereas

dynamic analysis is typically used for in-depth examination of malware. My goal is to improve dynamic malware analysis by offering a new set of features that aid the detection of malware.

Dynamic analysis is valuable in today's modern defenses against malware because it can provide information about malware behaviors that are complementary to static analysis. Dynamic analysis is used widely for producing sandbox reports, which detail actions executed by the program at run-time. Dynamic analysis uses the actions executed by the program, observed typically at an Operating System (OS)-level such as system calls, to build behavioral Indicators of Compromise (IOCs). These reports can be used by human analysts to understand what the malware is doing.

Although the current advances in dynamic analysis are helpful in modern defenses, there are still two problems that the current dynamic technologies fail to address. First, dynamic analysis is usually only run once per sample, which means dynamic analysis is based on the behaviors of a single execution path, which is often an underapproximation of a sample's total possible behaviors. This problem is exacerbated because current dynamic technologies that detect malware only focus on detecting how a program interacts with the OS as a method of monitoring behavior. This means they detect OS-level behavior, such as system calls or Windows library calls made by a program during its execution, and derive their IOCs based on the sequence of OS calls seen at run-time. I call these *external behaviors*. These external behaviors dismiss internal behavior, behavior that occurs in the program without the interacting with the OS, which I prove is valuable in enhancing a tool's ability

to reliably and robustly detect malware. Additionally, existing tools solely rely on OS calls or sequences of OS calls to determine malicious behavior. Existing dynamic analysis is adversely affected when these specific malicious sequences of OS calls are not present in the single instance of behavior analysis. Second, dynamic analysis has been heralded to defeat obfuscation and packing because executing the program can reveal the OS calls made by the program. However, current tools cannot actually detect the methods of obfuscation or unpacking. They simply bypass the malware's defenses to try to uncover the OS calls made by the program. They miss the methods of how the malware hid their code and thus sacrifice a potential way of further differentiating malware from goodware.

In order to increase existing dynamic malware detection tools' abilities to detect malware and address some of its deficiencies outlined above, I propose utilizing dynamic program-level (DPL) information to complement OS-level detection. DPL analysis is the study of a program's internal behavior at an instruction-level during execution. DPL analysis is fundamentally different from OS-level analysis in that it focuses on how the program is behaving within itself, rather than how it interacts with its environment. This leads to a class of behaviors that are orthogonal to OS behavior and thus complementary to existing dynamic analysis. I show that DPL information, when added to dynamic OS-level analysis, has the ability to increase the detection of malware and analyze a class of behaviors that OS-level analysis cannot.

I have built a malware detection tool that utilizes machine learning with a comprehensive feature set including my novel DPL features. To prove that DPL

features are capable of improving existing dynamic analysis technologies, I detected two classes of DPL information: *potentially evasive* and *general program execution.* I define DPL potentially evasive actions as any dynamic program-level behavior that a program, malicious or benign, uses to hide the true code that is executed. I detect the following eight DPL potentially evasive classes of behavior: *self-modification, dynamically-generated code, section-related obfuscation, unconditional to conditional branch conversion, overlapping code sequences, hidden functions, dynamic import address tables (IATs), and call-return obfuscations.* I also detect three classes of general DPL information, which I define to be features that, although collected at an instruction-level, can quantify a program's dynamic execution characteristics. The three general DPL classes I detect are: *function analysis, control flow analysis, and instruction execution.*

DPL features complement dynamic OS-level features in the following ways. First, it is gathered from monitoring the instruction sequences executed within the program. The use of DPL analysis gives access to wealth of information that has not been used in previous literature for the purpose of malware detection. I show and explain in this section how DPL information is orthogonal and distinct from OS-level features. Second, examining DPL information enhances dynamic analysis's ability to correctly detect malware when the monitoring of OS-level calls produces little to no behavioral signatures. DPL features can be more fundamental to the nature of the program and my work shows that the information obtained, when combined with machine learning, can provably increase the detection rate of a OS-based tool.

Some of the DPL features above are known to the security community as be-

ing more associated with malware than with benign programs. However my work is the first to identify a set of automatically detectable DPL features that can be used to improve dynamic malware detection. My contribution is unique for the following two reasons. First, I prove that I can extract and quantify DPL-related methods that malware uses to specifically hide code to avoid detection. The class of DPL potentially evasive behaviors I detect specifically targets methods in which malware tries to avoid existing analysis techniques. The alternative for malware to not using these methods would be to be caught by other simpler detection techniques. Though my list of these behaviors is not exhaustive, it is substantial enough to prove that these class of behaviors are quantifiable and useful for the purpose of malware detection. Second, I show that with my addition of general DPL information, irrespective of first classifying these DPL features as malicious or benign, that machine learning can learn the differences in the way malware versus goodware execute at an instruction level, leading to a higher accuracy rate.

Additionally, my DPL features have a significant advantage over previous works that used features sets such as strings to detect malware. Malware can alter their bytes or strings arbitrarily to match those of benign programs, which can cause a tool to generate a benign-looking feature set for a malware sample. This inevitably will cause it to be classified falsely as benign. My potentially evasive DPL features are unique because they detect how malware tries to hide code. Statistically speaking, my testing shows that benign programs perform potentially evasive actions significantly less than malware. Thus, if malware tried to bypass my DPL feature detection, they would lose the ability to hide its malicious code and could be

caught by other analysis tools. This is a significant upgrade in comparison to other feature sets because the alternative to avoiding my detection is to be caught more easily by other standard methods. My general DPL information features provide additional value because they measure at an instruction-level how malware execute. In order to avoid detection from my general DPL information features, malware would have to start behaving at an instruction level like benign programs, which significantly limits malware writers.

My DPL analysis differs from other previous work for the following reasons. First, works such as [45, 46] focused on instruction-level analysis, but in a static deployment. Their technologies did not execute the program and thus could not observe the types of behaviors my tool collects. Second, works such as [22, 25] have analyzed program-level behaviors dynamically, but only in a forensic setting. Some of my behaviors, better explained in Section 3.4.2, have been studied for the purpose of better understanding malware behaviors. However, my contribution and novelty stems from my insight that these behaviors that occur at a program-level can provide a non-redundant set of information to improve existing dynamic malware detection tools. My work here shows a comprehensive evaluation of my feature set and shows that it is capable of reducing the percentage of malware missed by a dynamic detection tool. Third, although it is well known that dynamic analysis can defeat some obfuscations and packed malware, existing works such as [47] that claim to defeat these malware techniques to avoid static detection only use dynamic analysis to bypass them. These works do not use the presence of these behaviors or categorize them in any useful way for the purpose of malware detection.

I show in my results the following findings. First, I show that adding my DPL feature set to a dynamic OS-based machine learning malware detection tool reduces the percentage of missed malware, also known as the false negative rate, by 39.45%. Second, I show that my DPL features when added to a dynamic OS-based malware detection tool detect more zero-day malware than the latter by itself. Third, I show that in deployment scenarios, where the testing set is obtained independently of the training set, my feature set improves a malware detection tool's robustness. Last, I show that my DPL information is critical in improving the performance of an OS-based malware detection tool in cases where a malware's execution produces little to no OS behavioral signatures.

I show the impact of the program-level features on my tool's detection rate on a dataset of over 400,000 real-world programs, distributed roughly equally between malware and goodware. I tested a neural network multi-layer perceptron (MLP) machine learning model and show that the model's performance improvements when the DPL features are added. Overall, with my DPL features incorporated, my malware detection system can correctly detect 98.45% of malware while maintaining a false positive rate of 1%. My contributions in this section are as follows:

- Prove the addition of DPL features reduces the false negative rate of OS-based malware detection tool by 39.45%.
- Provide comprehensive analysis of my DPL features, showing that they are both complementary and orthogonal to OS behavioral signatures.
- Show that with DPL features, I can produce a malware detection tool that can correctly detect 98.45% of malware while falsely detecting only 1% of

goodware.

- Produce a publicly shared dataset comprised of over 400,000 Windows executables with 223,417 malware and 195,255 goodware that are unique according to their SHA256 hash.

The rest of the section is organized as follows. Section 3.2 gives a background into malware detection using machine learning. Section 3.3 covers prior works' feature sets. Section 3.4 covers my feature set in detail. Section 3.5 covers my dataset, specific implementation, and tools used to build my system. Section 3.6 shows my findings. Section 3.7 discusses this work's limitations and possible remedies.

## 3.2 Background of Machine Learning in Malware Detection

Machine learning is critical in a scalable malware detection tool. I believe there are three core components needed. First, a large, diverse dataset must be used in developing an accurate and reliable machine learning malware detection tool. Second, the detection tool must be able to handle all types of malware, including obfuscated and packed malware. Third, a comprehensive, robust feature set including multiple sources of information must be used. My aim is to meet each of these requirements to produce a highly accurate, robust detection tool.

A machine learning-based malware detection tool is built in the following way. First, a labeled training set is obtained to train the classifier on what previously known goodware and malware looks like. After the training phase, a testing phase occurs where new unlabeled data is sent to the machine learning classifier for it

to classify as benign or malicious. These predictions are compared against the true labels of the test set to measure accuracy. Typically, a classifier is measured according to various metrics, the two most important ones being the false positive rate, and the true positive rate. In the malware detection field, a true positive is when a malicious program is correctly classified as malicious. A false positive is when a benign programs is detected as malicious. The false positive rate is supremely important because a majority of real-world network traffic is benign. Thus, my goal will be to minimize the false positive rate, while maximizing the detection rate.

## 3.3 Existing Works' Feature Sets

Feature selection is crucial to a machine learning algorithm's efficacy in detecting malware. The features chosen quantify a sample into something a machine learning algorithm can understand. Thus, choosing how to quantify a program, in my case, into a set of features is vital in producing a good detection tool. The goal of a good feature set is to accurately quantify a program, ideally with features that differ in value between malware and goodware.

### 3.3.1 Static analysis

Prior work has largely leaned on static analysis to detect malware. However, as [6] showed, static analysis has drawbacks that can be mitigated by dynamic analysis. Neither static or dynamic analysis is a substitute for the other; rather they have different strengths and are used for different purposes. My goal is to

improve dynamic analysis given that it is also widely used in sandboxes to produce dynamic reports on malware behavior. Below, I cover prior work that used static program-level information to detect malware. However, my work is fundamentally different from these because my work is focused on complementing existing dynamic malware detection technologies by incorporating DPL information.

These papers [21, 45–63] focused on using n-grams, opcodes, or strings individually or in some combination as a feature set for machine learning and malware detection. Although these features can be extracted from instruction or byte level, they do not quantify program behavior like my feature set. The dynamic information I collect from program-level features has no overlap with the features typically used by static tools to detect malware.

My DPL features differ from any static analysis tool because my features are discovered through the actual execution of the program. Although some of my features explained below, such as self-modification, combine a version of static analysis with dynamic analysis to detect the behavior, my detection schemes can not be replicated with static analysis alone. Runtime behavior is key in accurately and reliably detecting DPL behavior that aids the detection of malware.

### 3.3.2 Dynamic Analysis

Dynamic analysis of malware has been heavily studied in the past. However, I show below why existing dynamic analysis of malware has focused on goals or methods that are distinct from ours.

These works [15, 30, 64–66] utilized dynamic OS-level behavioral analysis to detect malware. Dynamic OS-level behavioral detection focuses on monitoring the Windows Application Programming Interface (API) calls and system calls made by the program during execution. This type of dynamic analysis solely focuses on a program's external actions. Previous work has shown success in detecting malware with using this type of dynamic analysis because the external actions often signify some action or request a program is making to possibly inflict harm. However, this class of dynamic analysis differs from my work because my DPL feature set focuses on internal actions and behaviors that an OS-level dynamic tool cannot detect or monitor. I show in my results that DPL information is indeed complementary and distinct from OS-level information, which is ultimately proved in my feature set's ability to boost the performance of a malware detection tool.

The following works [22, 23, 25, 67] looked into dynamically analyzing malware at a instruction-level, but only for malware understanding – not malware detection. Work like [25, 67] only studied obfuscations in malware such as self-modification or dynamically generated code. Other works such as [22, 23] used dynamic analysis to detect unpacking to uncover potentially malicious code to analyze further. Although these works focus on utilizing DPL analysis similar to ours, their goals are entirely different. First, their goal is to detect obfuscations or unpacking for the purpose of understanding malware better. Theirs works did not study the presence of obfuscations or unpacking in goodware, which limits their work to understanding malware and not being a general malware detection tool. My DPL analysis, on the other hand, uses the presence of obfuscation or unpacking at a program-level to

differentiate malware from goodware.

[68] is unique in that the authors focus on automatically synthesizing discriminative significant behaviors to detect malware. However, their tool still focuses on OS-level calls. Although their work proved to be useful in automatically generated discriminative dynamic OS behaviors, I believe that my work would still benefit theirs because my tool focuses on internal behaviors that are orthogonal to their external behavior analysis.

[69] generates behavioral models that represent families of malware. These models are generated by analyzing which dynamic OS calls are made as well as an instruction-level dynamic taint analysis. Although their tool does incorporate some level of DPL information, my work is still distinct from theirs for the following reasons. First, their DPL analysis did not focus on detecting obfuscations in malware as a method of detecting malware. Rather, they use DPL analysis to build a behavioral graph, which was used to represent the malware. Second, the authors only generate behavioral models for six distinct malware families. Their testing, which involves less than 500 samples total, only proves that their method could work as a way to classify unknown programs as one of these 6 malware families with a 93% success rate. Thus, their method is unproven for benign behavioral models and it is unknown whether their tool would work as a general malware detection tool.

The only work that has looked at DPL obfuscations as a method of detecting malware is [70], where six obfuscations were detected in a 100K malware and 6K goodware. [70] had to manually determine discriminating features so that the obfuscations detected were not found in benign programs. Moreover they did not use

machine learning in their method, and used a series of ad-hoc heuristics. Their work showed it is possible to use the presence of obfuscation as a method of detecting malware, but only in a limited, non-scalable way.

## 3.4 My Feature Set

In total, I have 1,168 unique features, of which 454 were the number of times each opcode found in my dataset executed dynamically. However, I found that training with all the opcode features in my machine learning algorithm took significantly more time and decreased my overall accuracy slightly. Thus, I use Principle Component Analysis (PCA) to reduce the opcode feature subset. This left 714 features collected plus the 250 components from the PCA of the opcode subset, which I found to be an optimal choice. The general DPL information along with the dynamic opcode counts are referred to as DPL statistical features. All of my features were either boolean features or value features. Boolean features had values of one, if the detection was present, or zero, if it was not. Value features were detections that could not be represented in a binary value such as the number of total instructions that were executed.

### 3.4.1 Incorporation of Existing Features

#### 3.4.1.1 Static Features

I collect static information such as PE header information and section-related information like section entropy. These features are input into my feature set as

either boolean features or value features (those which take a range of float or integer values). For features such as language, I use a boolean feature per language to show which language the program is in. For features such as entropy, I use the entropy value as my feature.

The following sets of static information were incorporated in my static feature set as boolean features (present/absent): YARA rules, language, and packer type (using PEiD). The following sets of static information were incorporated in my static feature set as value features: number of static imports, static size of program, and number of sections.

#### 3.4.1.2 Dynamic Features- OS Features

I collect OS-level dynamic information such as when a HTTP request is made or a file is created. I utilize an OS-level tool to monitor the program's interactions with the OS, then use the calls made to generate dynamic behavioral signatures. These dynamic signatures are generated based on a single or series of OS-level actions made by the program. An example of an OS-level feature based on a single action is a HTTP request signature which is generated when a program makes a HTTP request. An example of an OS-level feature based on a series of actions is a API spamming signature which is generated when a program repeatedly makes the same API call in order to delay analysis. These signatures are used as binary features in my feature set. I do not manually assign any feature weights to my features. The complete OS feature set can be found in the Cuckoo Sandbox Monitor Repository [37].

### 3.4.2 Novel Dynamic Program-level Features: Potentially Evasive

The novelty in my work comes from my use of DPL information for the purpose of malware detection. I collect two different sets of DPL features. First, I collect DPL potentially evasive behavior (referenced as EV in Section 3.6) features. Second, I collect general DPL execution behavior (referenced as STAT in Section 3.6) features. I collect DPL features using a dynamic binary instrumentation (DBI) tool. A DBI tool can monitor and collect information about each instruction in a program during its execution.

The premise of using DPL features is that these features can help provide a more complete picture of a program's execution and give a fine-grained look into the program's behavior. Specifically in the case of malware, traditional OS-level detections have relied on the presence of specific OS calls. However, OS calls alone typically only capture malicious behaviors such as installing a keylogger. My potentially evasive DPL features focus on capturing how the program internally arrived at executing its code. All of my potentially evasive DPL features focus on detecting when a program obfuscates or attempts to hide the code that is executed at run-time from static analysis methods.

In the case of my DPL potentially evasive detections, I detect DPL behavior that I believe is more prevalent in malware than in goodware. After analysis of malware at a program-level, I can leverage the fact that malware actively attempts to avoid detection of static and OS-level tools. Goodware typically does not. My DPL potentially evasive detection schemes focus on how malware tries to avoid triggering

static and OS-level tools' alerts. Using this distinction, I are able to create a tool that detects a wide variety of potentially evasive DPL behaviors that are fundamentally different in malware than in goodware. Some of these DPL behaviors can occur in both malware and goodware, but it is my belief, as justified by my results, that they occur more often in malware. Additionally, these DPL features cannot be detected by existing OS-based dynamic tools. Thus, my DPL detections provide an orthogonal set of information that is more prevalent in malware than in goodware, ultimately increasing an OS-based dynamic malware detection tool's accuracy.

My tool detects self-modification, dynamically-generated code, section-related obfuscation, unconditional to conditional branch conversion, overlapping code sequences, hidden functions, dynamic IATs, and call-return obfuscations. In total, I detect eight mechanisms that malware I studied uses to avoid detection from static and OS-level tools. Below, I describe in detail how each of these obfuscations are detected.

#### 3.4.2.1 Self-modification

**Description**: Self-modification is the process in which a program modifies its existing code during its runtime and then executes the new code. Self-modification occurs dynamically, which means the code that replaces the existing code may not be discoverable statically. Self-modification can occur through changing the read/write/execute permissions on a section of code, to later overwrite and execute it. Malware often use self-modification to hide malicious segments of code from

analysis tools.

**Implementation**: Self-modification detection is not novel, but the use of its presence and its varying levels of implementation for the purpose of detecting malware versus goodware is. I not only built my tool to generically detect self-modification (like most prior work), but to also detect the self-modification of op-codes, operands, and self-modification in dynamically generated code individually. I did this for two reasons. Previous work [70] showed that there are differences between obfuscations that occur in malware and obfuscations that occur in goodware. By detecting self-modification in varying degrees, my analysis tool helps the machine learning algorithm learn these patterns. Additionally, I found that the more data points I gave to machine learning to quantify programs, the better it performed. Thus, I leverage a DBI tool's ability to obtain specific DPL behavior that otherwise could not be obtained to enhance my DPL feature set.

I detect self-modification as follows. I maintain a dynamic disassembly as the program executes. A dynamic disassembly, as I define it, is built by using recursive traversal to build a limited static disassembly. Recursive traversal is a disassembly technique which starts at a target of an indirect Control Transfer Instruction (CTI), continues to the targets of direct CTIs and ends at a set of indirect CTIs. The disassembly process is continued when the target of the indirect branch is determined (right before it executes). While monitoring the execution of the program, my tool checks to see if the current address matches a previously disassembled address. If the current address has been seen before, my tool compares the current instruction bytes to the previously disassembled bytes. If they do not match, my

tool logs which portion of the bytes do not match (opcodes, operands) and outputs the corresponding boolean feature.

**Robustness**: This detection is hard to evade if the malware tries to use self-modification. Since I detect the self-modification of both opcodes and operands, my comprehensive analysis disallows self-modifying malware from going undetected. My tool constantly analyzes the code executed at an instruction level, the only way to avoid detection would be to not self-modify.

My detection implementation does detect some self-modification instances in goodware tested. However, my testing shows the number of occurrences is significantly lower in goodware versus malware. Additionally, because my tool does not use the sole presence of self-modification as an indication of malware, but rather relies on machine learning to use statistically significant trends, having a small number of self-modification detections in goodware does not greatly diminish this detection's efficacy.

#### 3.4.2.2 Dynamically generated code

**Description**: I define dynamically generated code as the process of allocating new memory, writing code to that memory, and then executing that code. Malware can use dynamically generated code to create and run code at runtime that may not be discoverable statically. My definition of dynamically generated code differs from my self-modification detection because dynamically generated code occurs when newly allocated memory is written to then executed. Self-modification occurs when

existing code in the program's code image is overwritten then executed.

**Implementation**: Although the detection of dynamically generated code is not novel, the use of its presence as a method of detecting malware is something that has not been studied outside of [70]. Similar to my self-modification detection scheme, my tool detects varying ways dynamically generated code occurs in malware I studied in practice.

My detection scheme for dynamically generated code detects the following as boolean features. The first feature is a boolean value detecting whether any dynamically generated code is executed. The second feature is whether virtual memory is written to without the use of standard Windows API calls like memcpy and then is executed. This feature specifically targets malware that try to avoid using standard Windows APIs to write to memory and instead use, for example, the x86 mov instruction to write to memory. The third feature is whether data is copied from a data section in the program to virtual memory then executed. This feature targets malware that may hide its malicious code in its data sections, which can be encrypted to harden its defenses against static analysis tools. Only during its execution is the malicious code unpacked and executed. The last feature is used to signify code cache behavior. Code cache behavior is when a program repeatedly writes and executes code in allocated memory. This is another way malware can unpack itself or hide its code from being statically discoverable.

**Robustness**: Because these detections occur at the program-level, malware has a significantly harder time evading these detections. Malware can choose not to dynamically generate code, but then loses one major method of evading static

detection and creating code at run-time. Previous OS-level approaches can only monitor memory writes that occur if the program uses a Windows API call or OS call. However, programs can directly write to memory locations using a mov assembly instruction. My DPL tool has the capability to detect these memory writes ensuring accurate and reliable detection.

Goodware can also dynamically generate code for benign reasons. My analysis shows that varying degrees of dynamically generated code did occur in both goodware and malware. However, the prevalence in malware was much higher than in goodware, especially for the last three features described in the previous section.

#### 3.4.2.3 Section-related obfuscation

**Description**: Section-related obfuscation involves misusing section labels or permissions in order to hide code or confuse disassemblers. Malware can initially mark sections of its program as read/write to make it look like data. However, it can later change the permissions of specific address regions within that data section to read/execute so that it can execute code. This obfuscation allows malware to hide code in sections initially marked non-executable, which may not be analyzed by static analysis tools.

**Implementation**: My tool detects the following section-related obfuscation features. The first boolean feature notes whether any of the sections have read, write, and execute permissions. This set of permissions allows programs to write and execute arbitrary code within its sections. This can be used for malware to self-

modify and dynamically generate code. My tool uses DynamoRio's internal API to determine the permissions of each section in the program when it is loaded into memory. The second boolean feature is output true if an instruction executes from a section of the program that was initially marked as non-executable. When the program is loaded into memory, my tool saves the address ranges of each section of the program to later determine if an instruction is executing from a non-executable subsection.

**Robustness**: My constant instruction-level analysis makes it nearly impossible for malware to use this obfuscation without being detected because my tool is able to track all executed instructions. My tool does not detect the changing of permissions explicitly, but can guarantee that if malware tries to execute an instruction from a region initially marked as non-executable, it will be detected.

#### 3.4.2.4 Unconditional to conditional branch conversion

**Description**: Unconditional to conditional branch conversion occurs when a program converts an unconditional branch to a conditional branch that always evaluates to only one of the fallthrough or target address. This conversion is used by malware to introduce more control flow paths or place junk code on the unused path to confuse a disassembler. By creating more code paths or invalid disassembly at the unused path, malware can hope to either create more work for a reverse engineer or potentially cause errors in the disassembly process.

**Implementation**: My tool detects this DPL behavior with a boolean feature

as follows. At every new conditional branch found, my tool disassembles the next basic block at both the target and fallthrough address. If either basic block has an invalid instruction, this boolean feature is output true.

**Robustness**: It is nearly impossible to determine if a conditional branch always evaluates one direction. Thus, although this is not a guaranteed method of detecting all instances of this conversion, I felt that it was a good compromise between a more complicated implementation, which would determine whether theoretically a conditional branch will always evaluate in one direction, and a simplistic implementation, which marked every conditional branch that only evaluated in one direction during a program's execution as malicious. My implementation does not detect if malware only uses this obfuscation to introduce more code paths, but does detect if malware places invalid instructions at either code path with high reliability. This detection can only be done at an instruction level.

#### 3.4.2.5 Overlapping code sequences

**Description**: Overlapping code sequences are unique code sequences existing within the same set of code bytes. Because the x86 architecture is a complex instruction set computer (CISC) architecture with variable length instructions, unique instruction sequences can exist at different offsets within the same set of bytes. Malware can use this obfuscation as a method of hiding code within sequences of other code to avoid static and OS-level detection.

**Implementation**: My tool detects this behavior by maintaining a dynamic

disassembly of the program, as described above, and dynamically checking to see if the program is executing an instruction that begins in the middle of an already disassembled instruction. To differentiate my self-modification detection and overlapping code sequences, my tool checks to ensure that the current instruction's bytes exactly match the previously disassembled bytes, with the only difference being the address at which the new instruction sequence begins. If my tool detects that the program is executing an instruction at an address that is in the middle of an already disassembled instruction and the current instruction bytes match the bytes of a previously disassembled region, my tool outputs this feature.

**Robustness**: This DPL feature cannot be monitored using only OS-level analysis because a disassembly of previously seen instructions as well as the current instruction executing must be kept. My use of a DBI tool ensures my tool will detect this behavior if it occurs dynamically and makes it impossible for malware to use overlapping code sequences to hide code without being detected. The only way malware could avoid my detection is to not use this obfuscation. The only way a static tool could detect this obfuscation is to disassemble the code of the program at every byte offset, which will produce a high number of incorrect disassembly traces and a high number of unexecuted code paths.

#### 3.4.2.6 Hidden functions

**Description**: Hidden functions are functions in a program that are not called by the traditional call instruction. In x86, a call instruction combines the function-

ality of a push and jump instruction. However, in order to avoid or deter analysis schemes, malware will use a manual push and jump instruction in place of a call instruction. This can help the malware hide functions from analysis tools that target the presence of call instructions to identify functions.

**Implementation**: My tool detects this behavior through two features, a boolean and value feature. The boolean feature denotes when at runtime, a return instruction goes to the instruction immediately after a jump instruction. This tells my tool that most likely, this jump instruction was used in place of a call instruction. The value feature detects how many times this occurs. My tool does not actively track push then jump instruction combinations because malware can place an arbitrary number of instructions in between a push and jump to evade this detection.

**Robustness**: Although my method of detecting hidden functions is not a guaranteed method of finding all hidden functions in a program, I felt that this was a good compromise that detected instances of this obfuscation without implementing a complicated, heavy detection scheme. My scheme resulted in a non-trivial number of detections in my malware dataset and disallows malware from utilizing a push/jmp instruction in place of a call instruction to avoid function-based detection schemes. The use of a DBI tool allows the detection of this obfuscation that otherwise would not be possible.

#### 3.4.2.7 Dynamic IAT

**Description**: The import address table (IAT) is a list of functions that a program imports during run-time. Static analysis methods have been shown to be able to detect malware based on its imports [71]. To counter this detection, malware can bypass the IAT entirely at run-time to avoid prepopulating it during compilation. At runtime, malware can dynamically calculate the address of functions it needs in order to call them directly without having to use the offset located in its IAT. This allows malware to avoid giving up information statically about what functions it may use during its execution.

**Implementation**: My tool detects this obfuscation by measuring how many of the program's function calls were located in the IAT. After studying malware and goodware, I determined that having a feature that noted whether or not more than half the function calls made by a program bypass the static IAT was a good method of finding this obfuscation. My tool monitors all function calls made by the program dynamically, and compares them to the initially loaded IAT.

**Robustness**: Because my tool analyzes the initially loaded IAT and then continuously monitors the function call made by the program, my tool is able to detect all instances of malware trying to call functions that were not listed in the IAT statically. Additionally, since my tool also detects hidden function calls as outlined above, malware has two significant obstacles in the way of hiding dynamic function calls. The only way to avoid this DPL feature is to statically list all functions called in the IAT. My determination to use 50% as the benchmark to determine if this

obfuscation was present was based on data from goodware and malware's behavior. I saw that some instances of goodware may call functions not listed in the IAT, but it was very rare for goodware to have more than 50% of its function calls made not in the IAT. Goodware has no clear reasoning to do this, while malware has more motivation to use this obfuscation, which my tool leverages.

#### 3.4.2.8 Call-return obfuscation

**Description**: Call-return obfuscations involve misusing the call and return instructions to confuse analysis tools on what the real control flow paths are. In particular, the return instruction is effective because it is an indirect control transfer instruction (CTI). Static tools cannot decipher the control flow of a program after an indirect CTI because the target of the CTI is typically only known at run-time. Malware often times abuse the return instruction because it can conceal the control flow of its execution.

**Implementation**: My tool detects call-return obfuscations by several methods. First, it keeps track of all calls and returns made by the program and makes sure that returns match up with calls. If they do not, my tools outputs that as a boolean feature. My tool also outputs as a value feature the number of times a return instruction goes to an invalid location (i.e. a location that did not have a corresponding call instruction immediately before it).

**Robustness**: My tool's detection here is based on the correct usage of a return instruction (meaning all returns go to the instruction after the corresponding call

instruction). My testing and analysis shows that return instructions do not always go to a corresponding call instructions in either goodware or malware. However, my testing also showed that this occurred a significantly higher number of times in malware than in goodware. Thus, my detection is simple as explained above, which disallows malware from abusing the return instruction. The only way malware could avoid my detection or behave similarly to goodware is to very rarely use the return instruction to obfuscate control flow, which greatly diminishes the utility of this obfuscation, or not use it at all, which again eliminates one method of avoiding detection from other tools. OS-level tools miss this behavior because they cannot monitor call or return instructions.

### 3.4.3 Novel Dynamic Program-level Features: General DPL Execution

In addition to the possibly evasive detections, I collect general DPL execution statistics. Although these detections are not as targeted as my obfuscation DPL features, my belief, which is later confirmed with my results, is that these sets of data differ between malware and goodware. With the level of specificity a program-level tool is able to provide, I show the addition of this set of features helps my machine learning detection tool differentiate malware and goodware more accurately. I found that earlier works had success incorporating features that were not clear indicators of maliciousness such as the opcodes found or a program's bytes. Thus, I incorporate the following sets of DPL data as value features that help quantify how a program

behaved at an instruction level that existing OS-based behavioral detection tools do not capture.

#### 3.4.3.1 Function analysis

In addition to detecting hidden function and call-return obfuscations as described above, my tool also tracks generic function execution details. My tool outputs as features the number of external functions found (functions that are located outside of the program code), number of internal functions found (functions that are located inside of the program), and the number of times internal and external functions are called dynamically.

I hypothesize that this set of DPL information can give a higher-level view of the control flow and call graphs of the programs studied. I found that malware I analyzed typically relies less on external dlls and dependencies in order to ensure that it can execute regardless of its environment. This is backed up by my higher successful run rate in the dynamic testing malware versus goodware. Thus, malware may call more internal functions than goodware, which this set of information aims to quantify.

#### 3.4.3.2 Control flow analysis

I use DPL analysis to quantify the control flow of a program. I detect the number of edges between basic blocks in the program's execution. I gather as value features the number of outgoing edges from basic blocks that end in an indirect

branch (such as returns), and the number of times indirect branches execute. I also output the number of times the indirect branches go inside or outside of the program's code sections. These metrics can be useful in detecting code flattening, which is a process in which a malware jumps to a majority of its code from a single basic block so that its control flow graph is essentially flat. Malware employ this to prevent static tools from building full control flow graphs.

#### 3.4.3.3 Instruction execution

My tool monitors and quantifies the overall execution the program under analysis. It measures the following statistics: the average number of times a basic block executes, the max number of times a basic block executes, the number of unique opcodes that execute, and how often each opcode is executed dynamically. All of these are output as value features. I saw in previous works that using opcodes as a feature set performed well; thus I felt that analyzing the opcodes that executed dynamically would be useful. My results show that by monitoring the general execution trends of goodware and malware, machine learning is able to decipher statistically significant patterns that ultimately improve its accuracy by a non-trivial amount.

My results show that my highest accuracy was obtained when all of my DPL features were incorporated into my model. My primary goal was to show that the DPL information, whether it measured obfuscation-related behavior or statistical information, provides a non-redundant set of information that is orthogonal to existing dynamic detection tools. By leveraging machine learning, similarly to previous

works before me, I was able to incorporate these statistical features that were able to quantifiably boost the performance of my tool because it provided additional instruction-level insight that further differentiated malware from goodware.

Ultimately, the novelty and usefulness of my work comes from these DPL features. I show in my results that these features are able to provide an non-redundant set of information to existing dynamic OS-level tools that helps differentiate malware from goodware. My results prove that not only are my DPL features useful for better understanding malware's behavior, but also critical in reducing the number of malware missed by OS-based tools.

## 3.5 My Implementation

The specifics of the tools used for my implementation and experimental setup are described below.

### 3.5.1 Analysis Overview

**Cuckoo**: I utilized Cuckoo Sandbox [37], which is an open-source sandbox manager for automated dynamic program analysis. After a program is submitted to Cuckoo through its API, it is uploaded to a virtual machine (VM) and executed. During the program's execution, it is monitored using Cuckoo Monitor, which is Cuckoo's OS-level monitoring tool. It captures all OS-level behavior and then generates behavioral signatures that are output into a Cuckoo report. The Cuckoo report also contains static information that is used in my feature set as well.

I used a Windows 7 environment for my VM. Each VM is given 1/2 GB of RAM and 1 virtual CPU. Each sample was run in its own VM for one minute. I felt that one minute was enough time to allow the program to run, while keeping my testing time within reason. Cuckoo also has a module that mimics human interaction to simulate an endpoint. My VMs use a Tor gateway to give the program access to the Internet while obscuring where the request is coming from. Although there are safety and ethical concerns with allowing malware to connect to the Internet, because all the malware under study have been detected by antivirus tools and have been in existence for more than a year, I felt that it was an acceptable decision. In addition, [72] calls for real Internet connection for malware when analyzing it to ensure more substantial execution. I loaded the VM with common DLLs such as msvc.dll in order to maximize the number of applications that ran.

**DynamoRio**: I used a DBI tool called DynamoRio to perform DPL analysis [73]. DynamoRio is an open-source tool used to instrument binaries at run-time and monitor all of the instructions it executes. Using DynamoRio, I build custom DPL detections and statistics collectors that are added to the Cuckoo report at the end of its execution. These detections and statistics are used as a part of my DPL feature set.

### 3.5.2 Machine Learning

I utilized a neural networks-based approach due to number of features and samples I had. Specifically, I used a multilayer perceptron (MLP) neural network to

produce my malware detection tool. I saw that from preliminary testing, MLP was robust and generalized the best on all of my test sets. I also tested random forest and Support Vector Machine with various kernels, but chose MLP because of its performance. I are not claiming MLP is the best choice for malware detection. My goal here is not to extensively test different machine learning models, but rather, I aim to show how my DPL features can positively impact a high performing machine learning-based malware detection tool.

In my MLP configuration, I used one hidden layer with a size equal to the input layer with dropout set to 0.5 between each. I used a learning rate of 0.01, learning rate decay of 1e-6, and Nesterov momentum set to 0.9. For my first two layers, I used a rectified linear unit activation function. I used these parameters in order to help the model train quickly and learn efficiently. These are standard parameters and practices when it comes to using an MLP model. For the third layer, I used a softmax activation function to output the probability of maliciousness. In total, my largest feature set produced a model with 1.8 million trainable parameters. I trained my model and used a separate validation set to measure the learning progress. The test set was not used until all the model's parameters were set. I trained the MLP for 100 epochs because I observed that after 100 epochs, my tool's accuracy on my validation set leveled off at 98.7%.

To implement my machine learning detection tool, I used Scikit Learn [74] and Tensorflow with Keras [75]. I used Scikit to generate my feature matrix and used Tensorflow with Keras to implement my neural networks. I scaled my features to range from -1 to 1. Scaling my features ensured that all the value-based features

did not outweigh another. Normalization scaled the features to have unit variance. These are standard practices.

I trained and tested my MLP on a server that had a NVIDIA GeForce GTX 745 GPU.

### 3.5.3 Malware Detection Tool Implementation

In order to improve the implementation of my malware detection tool, I paired my MLP tool with ClamAV- a popular open-source AV tool [76]. I used ClamAV in parallel with my MLP malware detection tool to simulate a real-world deployment. My system worked as follows. If either ClamAV or my tool flagged the program as malicious, I considered it malicious. If neither considered the program malicious, I categorized the program as benign. I did this because ClamAV has an extremely low false positive rate (0.005% on my goodware dataset). Thus, the detection rate of my system improves without hurting the false positive rate.

This likely emulates how a real-world system works; they often pair all types of tools to build a comprehensive, high performing tool. More importantly, I show that my DPL features can still aid the detection of such a real-world tool and thus proves the importance of my tool to any malware detection tool.

### 3.5.4 My Dataset

I first collected a set of over 400,000 programs consisting of 223,417 malware and 195,255 goodware. These are programs that execute dynamically in a Windows

7 environment.

As with any dynamic analysis scheme, not all the samples tested executed in my test environment. Thus my 400K dataset was obtained after filtering out the ones that did not run successfully within my VMs. This was an inherent downside to using dynamic analysis, but I felt that by having a dataset with more than 400K that did run sufficed. I discuss possible remediations in Section 3.7.

**Malware**: My malware dataset was obtained from VirusTotal, who granted me a private key so that I could download these programs. Virus Total is a online database of programs maintained by Google that are queried against 50+ antivirus tools to determine maliciousness. I chose 5,000 randomly selected malware per year from 2001 to 2008. I then used 20,000 randomly selected malware per year from 2009 to 2017. Lastly, I used 40,000 randomly selected malware that had a PE timestamp that was before 2001 or after 2017. I assumed for these samples, the timestamp had been tampered with and thus categorized all of them together as unknown. I chose this distribution to ensure a diverse malware set, while weighting my dataset to more recent samples.

I chose to download PE32 programs that had at least 15 or more detections on VirusTotal to ensure that the malware being testing was indeed malicious. I chose 15 or more detections because it means that more than 25% of the queried anti-virus tools determined the program to be malicious. One of the main issues with other malware datasets, as [72] points out, is that some publicly available malware sets have no guarantees of being malicious. Rather, they may simply be samples of unknown programs. This is problematic because if the ground truth of the machine

learning dataset is unreliable, then the results may be unreliable. Thus I felt that only taking programs with at least 15 detections was a necessary step. Similar measures were taken in related works [77, 78].

**Goodware**: In order to obtain a goodware dataset that was large and diverse, I downloaded programs with zero detections on Virus Total. I felt that if a program has been in VirusTotal's database for more than a year with zero detections, then it was most likely a benign program. I saw the same filtering being used in related works such as [77, 78]. I downloaded all PE32 programs that had zero detections from 2005 to 2016.

Defining a benign program is a difficult problem. Simply put, a benign program can be seen as a program that does not perform any unwanted action. If 50+ antivirus tools deemed that a program did not execute any malicious actions, then I believe that this program is most likely benign. In addition, most other datasets used in prior work are dominated by installing a clean version of Windows and extracting all the system executables. However, this taints the benign set because it is coming from one source. Furthermore, there is no guarantee that the programs there will provide diverse examples of benign programs that may act suspicious. Including these types of programs in my benign set are important to ensure my tool's false positive rate is one that will generalize well even when deployed in the wild. Programs could be submitted to VirusTotal because the community wanted to check if it was benign or malicious. Thus, I believe that the programs that were submitted to VirusTotal and cleared by 50+ antivirus tools provide better examples of suspicious benign programs and ultimately provide a more robust, diverse

dataset.

## 3.6 Results

I used a standard ten-fold cross validation (CV) in order to obtain my results. This means I used ten folds (each fold consisting of 90% of the dataset for training, and 10% for testing) such that each sample was in the test set once. This is a standard practice in machine learning to ensure that the validity of the results.

### 3.6.1 MLP-based Malware Detection Tool Results

Table 3.1 below shows how my MLP-based malware detection tool performs when set at a 1% false positive rate (FPR). I show the improvements in the detection rate (DR) as I add to the feature set. The first feature set consists of all static and OS-level features obtained from the Cuckoo report (this set is labeled OS). The second feature set includes everything in the OS feature set along with the potentially evasive (EV) DPL features (this is labeled OS+EV). The third feature set includes the OS+EV features along with the DPL features that quantify general execution statistics (this is labeled OS+EV+STAT). I also show the F1 score, and area under the receiver operating characteristic (ROC) curve (AUC) for each configuration. The results shown below are for my ten-fold CV test on my dataset described in section 3.5.4 in conjunction with ClamAV.

From Table 3.1, it is clear that MLP's best DR, F1 score, and AUC are achieved when the DPL features are incorporated. Since the false negative rate decreases from

| Feature Set | DR w/ ClamAV | F1 score | AUC |
|---|---|---|---|
| OS | 97.44% | .9829 | .9974 |
| OS+EV | 98.00% | .9855 | .9980 |
| OS+EV+STAT | 98.45% | .9879 | .9981 |

Table 3.1: MLP Results with a 1% FPR

100 - 97.44 = 2.56% to 100 - 98.45 = 1.55%, that is a decrease of 39.45% when used the DPL features versus just the OS feature set. The false negative rate is the percentage of malware missed by a classifier and is calculated by subtracting the DR from 100%. The DPL features here add clear value in helping detect more malware. In addition, the F1 score is 29.24% closer to a perfect score and the AUC is 26.92% closer to a perfect score when using the DPL features are incorporated. The increase in AUC is important because it shows that MLP is able to obtain a higher DR across a range of FPRs, not just when operating with a 1% FPR. This is shown in the ROC plot in Figure 3.1, which shows the detection rate or true positive rate for when the FPR ranges from 0% to 5%.

To further study my DPL features' impact on the MLP-based malware detection tool, I analyzed the false negatives of the OS feature set versus the false negatives of the OS+EV+STAT feature set without the aid of ClamAV. A false negative is when a malware is incorrectly classified as goodware. When the MLP tool used the OS feature set, it missed 5,359 malware out of 223,417 (2.40%) when normalized to a 1% FPR. When the MLP tool used the feature set with my DPL features included, it missed 3,454 malware out of 223,417 (1.55%) when normalized

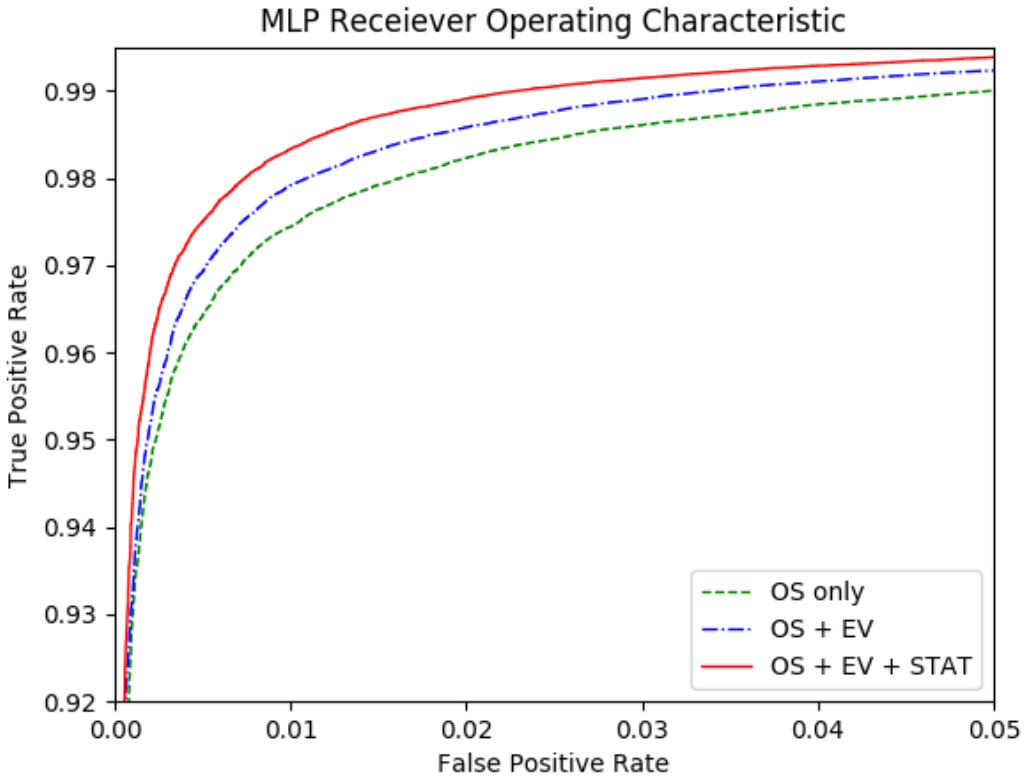


Figure 3.1: ROC plot for MLP with $0\% \leq \text{FPR} \leq 5\%$

to a 1% FPR. I analyzed the subset of 2,462 malware that the MLP tool was able to detect because of my DPL feature set. For these malware, I measured the average probability of maliciousness output by the MLP tool when using the OS feature set vs the OS+DPL feature set. On average, the MLP tool categorized the likelihood of this subset of 2,462 malware as 28.84% maliciousness. After the addition of my DPL feature set, the average likelihood of maliciousness was increased to 86.81% (an increase of 57.97%). This boost in probability of maliciousness shows that not only are my DPL features helpful in detecting more malware, but that they also provide information that significantly increases a MLP model's predictive power. For samples that my MLP model was more than 86.81% confident it was malicious,

it was 99.57% accurate. This demonstrates that my DPL features' impact on my MLP tool's accuracy is not a trivial improvement.

Analyzing feature importance when using a MLP model is an unsolved problem because MLP does not give feature importances. However, I did filter out a few DPL-feature sets by training and testing my MLP model with a four-fold CV and excluded one set of DPL-feature sets at a time. Although this is also an imperfect test of feature importance, I was able to eliminate DPL features that were not useful for increasing the accuracy of my tool.

### 3.6.2 Generalization Test

I conducted a small experiment to approximate how my DLP features affect a more realistic deployment scenario where the training set is collected independently of the testing set. I trained two MLP models, one based on the OS feature set and one with the OS+DPL feature set, on my entire dataset. I tested on two different sets collected from two reputable cybersecurity companies (unnamed for confidentiality purposes). The companies mentioned that both datasets were of advanced malware that were particularly hard to classify, that they used for comparative testing of tools. Malware set one (from company one) contained 340 samples and malware set two (from company two) contained 720 samples. Both sets were confirmed to be malicious by the respective companies. Each set of malware were collected completely independently and had no overlap from my training set. I used the same setup and threshold of maliciousness as my experiment above (covered in Section

3.6.1). The results are shown below in Table 3.2.

| Test Set | Feature Set | DR w/ ClamAV |
|---|---|---|
| Set 1 | OS | 85.59% |
| | OS+DPL | 91.18% |
| Set 2 | OS | 80.14% |
| | OS+DPL | 84.44% |

Table 3.2: Generalization Test Results

The results above show that my DPL features are useful in increasing the MLP-based malware detection tool's ability to detect more malware in a challenging deployment scenario. For set one and two, my DPL features decrease the false negative rate by 38.79% and 21.65%, respectively. This is important because it proves that my proposed feature set is able to increase an OS-based malware detection tool's robustness in the real-world where the samples coming into a network could be different than what the original machine learning model was trained on.

I expect the results from my tool are lower than my expected detection rates because both of these companies' sets are used by the companies to test other malware detection tools. In order to do so, the companies intentionally chose hard-to-detect malware, which is presumably why the detection rates are lower than for my full dataset of randomly selected diverse malware. Additionally, these sets are small and thus have limited value in adequately evaluating my system, but I felt that showing the results was still important for completeness.

### 3.6.3 Zero-day Test

I tested the effect of my DPL feature set on a simulated zero-day test. I trained two MLP models, one with the OS feature set and one with the OS+DPL feature set, on samples in my dataset that had a PE timestamp of 2015 or earlier. I then tested on goodware and malware from 2016 and 2017. This resulted in a test set of 27,402 goodware and 36,578 malware. The purpose of this experiment is to show my DPL features help a dynamic OS-based malware detection tool more accurately detect zero-day malware. The detection rate shown is for when the FPR is set to 1%.

| Feature Set | DR w/ ClamAV | F1 Score | AUC |
|---|---|---|---|
| OS | 82.83% | .9097 | .9835 |
| OS+DPL | 87.50% | .9424 | .9853 |

Table 3.3: Zero-day Test Results

Table 3.3 shows that the addition of my DPL features reduces the false negative rate by 27.20%. This experiment approximating a zero-day detection shows that my DPL features again prove to be useful for an OS-based tool to more accurately detect malware. This additionally motivates the usage of my DPL features in any deployment scenario to aid existing dynamic OS-based malware detection tools perform better.

The time-analysis results above are likely worse than expected because program behavior changes over time. Thus, training on an older set of programs and

testing on new ones often produces a worse result than expected. This is partially why zero-day malware are hard to detect. However, the goal of using DPL features is not to solve the problems associated with zero-day malware; rather, I show in the results that my DPL feature set increases existing malware detection tool's performance against zero-day malware.

### 3.6.4 Performance on Malware with Few Dynamic Signatures

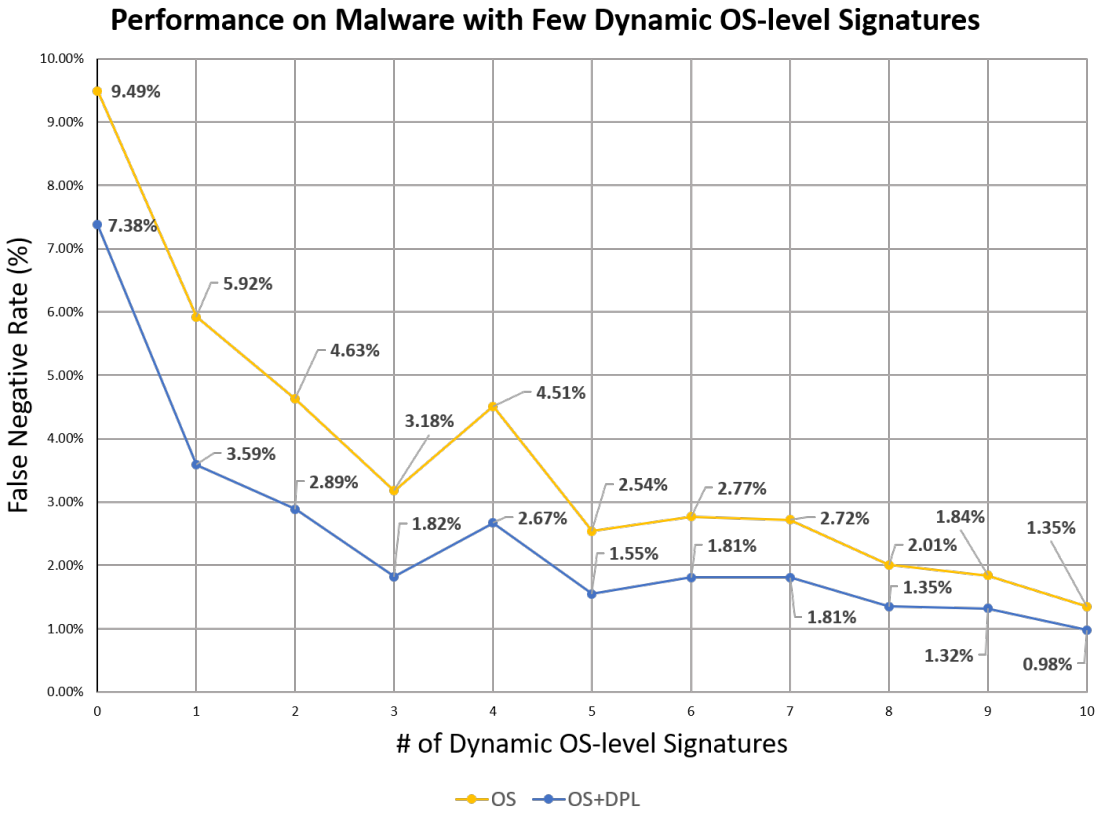


Figure 3.2: False Negative Rates of OS VS OS+DPL on Malware with Few Signatures

Often when dynamic OS-level tools are used, the dynamic signatures used to detect malicious behavior is reliant on a sequence of specific API calls being made.

For example, Cuckoo's keylogger OS behavioral signature relies on the execution of an API that registers an interrupt for every key pressed, then another API to write each keystroke to a file on disk. Malware detection tools that use OS-based behavioral detections rely heavily on these signatures in order to detect malware. As shown in Figure 3.2, I see that the false negative rate rises for as the number of dynamic signatures found decreases. In the cases where there are little to no dynamic OS-level signatures, existing dynamic tools have no other method of detecting these malware. The DPL features are able to decrease the percentage of missed malware by upwards of 43%. The use of DPL features is pivotal to detecting the malware that OS-level tools cannot because it captures a set of information orthogonal to OS-level malicious behavior.

With further analysis, I found that for the malware samples which the OS-based MLP model correctly classified, they had on average 8.88 dynamic OS signatures. For the malware samples which it got wrong, they had on average 5.05 dynamic OS signatures. Despite the reduced number of dynamic OS signatures, my DPL features are consistently able to reduce the false negative rate and maximize dynamic analysis' capability to detect malware.

## 3.7 Limitations

When performing dynamic analysis, having programs that do not execute due to dependency issues or are evasive are always concerns. In order to combat the dependency issue, I first downloaded standard libraries such as Microsoft Visual

Studio. Unfortunately, there is no feasible way to obtain all programs' dependencies to ensure execution. However, in typical real-world deployments this may not be an issue because a sandbox placed at a network's firewall will likely imitate the environment of the endpoints of the network. Thus, if the unknown program can execute on the endpoint, it will most likely be capable of executing in the sandbox.

Malware with anti-sandbox evasion is a harder, but orthogonal problem. Malware with anti-sandbox evasion is malware that uses techniques such as sleeping to avoid executing any significant behaviors unless specific requirements are met. Although I detect potentially evasive actions, I do not claim to defeat all evasive malware. The current advances in defeating it could be added to my tool in the future. However I collected my datasets without regard to anti-sandbox evasive malware, so I expect some of these malware deploy anti-sandbox evasion. Hence, all results in the section are valid for a collection of real-world malware, including evasive malware.

Adversarial machine learning is the practice of creating samples that are specifically crafted to trick machine learning models. In my case, adversarial machine learning could be used to create malware samples that have similar feature values to my goodware set, thus tricking my model into thinking malware are goodware. Although this is outside of the scope of my paper, I believe that adversarial machine learning would be more difficult to do when using my DPL features versus OS-level features only because my DPL features capture another class of behaviors. Malware would have to not only perform OS calls in a seemingly benign manner, but also execute their instructions in ways that are restricted to actions a typical benign

program would perform. Thus, my DPL features reduces a dimension of freedom malware has in terms of hiding its malicious code.

## 3.8 Conclusion

Dynamic program-level features provide a new frontier of information that is capable of aiding existing dynamic OS-level malware detection tools. Without it, dynamic OS-level tools miss internal behaviors that not only detail potentially evasive actions, but also provide additional useful information that can be used to detect malware despite the lack of OS-level behavioral signatures. My work shows that DPL features are key in producing a tool that can reduce the false negative rate of dynamic OS-level tools by nearly 40%, which can reduce an enterprise's cost of dealing with malware by 40%. My DPL features also qualitatively changes the way malware can behave because my tool detects and uses internal behavior as a method of detecting malware. My results show that in a variety of test cases, DPL features can further reduce the adverse impact of malware and present a future area of research in hopes to better defend against malware.

# Chapter 4: A Hybrid Static Tool to Increase the Usability and Scalability of Dynamic Detection of Malware

## 4.1 Introduction

Modern intrusion detection systems (IDSs) are responsible for stopping unwanted software from entering an enterprise's network, while ensuring that benign programs are not falsely flagged. IDSs that rely on static analysis can be deployed as active defense systems because static analysis is fast and can handle at scale the thousands of programs coming into the network. An active defense system is one that can stop programs in real time from coming into the network if flagged as malicious. However, active systems must maintain an extremely low false positive rate to ensure that benign programs are not falsely flagged. This is done by setting the threshold very high for how confident the IDS is in order to classify something as malicious, allowing some malware to slip through the detection system. Thus in order to deploy an active static IDS, the detection rate of malware is sacrificed at a cost of reducing false positives.

IDSs that analyze every file dynamically present a different strengths and weaknesses. Dynamic analysis is based on the behavior rather than the static at-

tributes of malware, which can make it more robust against packed, obfuscated, and previously unseen malware. Work such as [70] show that dynamic analysis can obtain useful information, which often leads to higher accuracies of detection, that static analysis cannot. IDSs that use dynamic analysis on every incoming file are typically deployed as passive systems, meaning that programs that arrive at the network are not blocked in real time. Because dynamic analysis requires the execution of the program in a protected environment, also known as a sandbox, analysis takes minutes to complete. Because malware is not blocked in real time, it is allowed to go past a network's defenses during analysis and wreak havoc before being potentially detected by the dynamic IDS a few minutes later. At this point, remedial action is taken at the infected endpoint to remove the malware and perhaps re-image the endpoint if the malware has run. I define the time elapsed between when a malware enters the network and is stopped as timeliness.

Most modern malware defense systems rely on hybrid approaches, which combine both static and dynamic analysis in order to defend against malware. However, I found from a literature and industry review that the way in which static and dynamic analysis have been combined is suboptimal. Most hybrid approaches work one of two ways. The first is a system which performs both static and dynamic analysis on all incoming traffic to obtain static and dynamic information in order to maximize a system's accuracy. The problem with this method is that although both sets of static and dynamic information are obtained, the computational and timeliness costs are prohibitively high because all programs are dynamically analyzed. The other primary hybrid approach is using a static analysis-based tool to

detect any incoming threats. The detected threats, based on static analysis alone, are sent to a Security Operations Center (SOC) to be analyzed further by other methods such as dynamic analysis. This method is problematic because it relies solely on the capability of static analysis to detect threats. As mentioned above, any active system maintains an extremely low false positive rate (to remain usable in the real world) by sacrificing its detection rate. Thus by only relying on static analysis, the malware detection system is incapable of defending robustly against all types of threats.

I present a new hybrid malware detection configuration that runs at the network entry point to an enterprise that strategically leverages the strengths of both static and dynamic analysis while minimizing their weaknesses.

My proposed detection system contains a two-step process. First, a tool which I call a *static-hybrid* tool analyzes all incoming programs statically and categorizes them into one of three buckets: very benign, very malicious, and needs further analysis. It is thus named to distinguish it from usual static tools, which categorize incoming programs into only two buckets: malicious or benign. The very benign bucket contains programs that the static-hybrid is highly confident are benign. This bucket contains near zero false negatives (*i.e.*, missed malware). The very malicious bucket contains programs that the static-hybrid tool is highly confident are malicious. This bucket contains near zero false positives (*i.e.*, benign programs falsely detected as malware). The needs further analysis bucket contains programs the static-hybrid tool is unable to make a strong determination on and thus is passed to the dynamic analysis tool, which is the second step of the process.

The programs located in each bucket provide unique improvements to existing IDSs. The programs in the benign bucket can directly bypass the dynamic analysis portion of my tool, reducing the computational resources needed for dynamic analysis. The programs in the malicious bucket can be blocked immediately, improving timeliness and reducing the need to conduct damage control after a program is identified as malicious in a passive IDS. The programs in the needs further analysis bucket are the only programs passed to the dynamic analysis tool. These programs, as my results will show, have a lower chance of being categorized correctly by a static analysis tool compared to a dynamic analysis tool. Thus by utilizing dynamic analysis on this subset of programs only, my tool's overall accuracy is higher than a strictly static IDS. In addition, because only the programs located in the very malicious bucket are immediately blocked, my static-hybrid tool is able to operate at a much lower false positive rate than a comparably built static-only tool, as my results will show.

My scheme reduces the computational resources by 27.5X in my salient configuration. This reduction is important for the practicality of a dynamic scheme. For example, a system that analyzes every file dynamically using a sandbox that encounters 200,000 files/day at an organization's network entry point (a typical file volume for a mid-size organization), I estimate would cost $24,000/year in cloud computing costs alone to run the full sandbox load, based on commercial cloud costs today. By reducing the computational load by 27.5X, the cost of protection would also drop by 27.5X with a very small loss in accuracy of protection.

In practice, any active system must maintain a near-zero false positive rate to

avoid adversely affecting an enterprise's users. Typical active IDSs block 100% of the programs flagged as malicious. However, my tool's two-step process allows it to only block a subset of programs flagged malicious. This is unique to my tool and I define it as the Blocked Benign Rate or BBR. The BBR is the percentage of benign programs stopped in real-time, which is distinct from the overall false positive rate that measures the total percentage of benign programs flagged as malicious. My goal is to produce a practical, scalable malware detection tool that strategically combines the benefits of both static and dynamic analysis. Thus, in order to ensure the practicality of my system, I chose my salient configuration to have a BBR of 0.08%. My results show that my salient configuration can still block 88.98% of malware immediately. Additionally, in practice I believe that the BBR can be further reduced by combining my system with other AV tools or file reputation websites.

My work differs from previous work because I are the first to use a static-analysis based tool to generate three answers instead of two. This allows my work to utilize the strengths of static analysis to quickly and correctly detect a subset of malware and goodware for which it is very confident, while using dynamic analysis to more accurately classify the remaining programs. I are the first to propose such a system to improve existing hybrid malware defense systems' scalability and performance. Because of my two-phase design, I can reduce computational resource requirements and improve timeliness of schemes that use dynamic analysis on all incoming traffic by much larger factors in comparison to previous work. I introduce a system that combines both static and dynamic analysis in an innovative way that

takes maximizes the strengths of both while minimizing their weaknesses.

My contributions are as follows:

- Propose a two-phase malware detection tool that uses a static-hybrid-based machine learning tool to reduce the computational resources needed, improve the timeliness, and reduce the alerts generated by dynamic analysis.
- Prove that on programs a static analysis tool has difficulty classifying, dynamic analysis can provide a dimension of information which enables it to more accurately classify them, ultimately improving my system's overall accuracy compared to static-only based methods.
- Show that my resulting hybrid system is able to reduce computational requirements of typical dynamic analysis by 27.5X, block 88.98% of malware in real-time with a 0.08% BBR (detecting 98.73% of malware eventually), and reduce the number of alerts generated by 9.5X.
- Obtain the results on my system by training its machine learning component on a set of over half a million programs, equally split between malware and benign programs (also known as goodware).

The rest of the section is organized as follows. Section 4.2 covers the related work in the field along with my distinctions. Section 4.3 covers my design and explains in detail the technology behind my scheme. Section 4.4 covers the specifics of my machine learning implementation, feature set, dataset, and testing setup. Section 4.5 covers my results.

## 4.2 Related Work

My goal has two main components. In comparison to dynamic analysis-based IDSs, my technology aims to reduce the computational load, while improving the timeliness of dealing with a malware. In comparison to static IDSs, my technology aims to perform more accurately. On these fronts, there are several related works that are explained below.

[79] had the most similar goal to ours of improving the efficiency of dynamic malware analysis, but tackles the problem through reducing the number of polymorphic samples tested. Polymorphic malware are malware that contain different bytes as to avoid signature-based detection, but perform the same actions. [79] states that due the high number of polymorphic samples, dynamic analysis is often unnecessary, thus it only dynamically analyzes each malware sample for a short period of time, then compares the dynamic analysis results with that of previous malware. If it matches, then analysis is stopped, thus saving time and resources. However, their work was only able to forgo 25% of analysis, while my work reduces the computational requirements by 27X. Additionally, spawning a sandbox takes a non-trivial amount of time meaning that their scheme cannot in real time stop malware (*i.e.*, there is no improvement in timeliness).

Another class of related works are papers that aim to detect malware similarity [80–87]. These works use static or dynamic analysis in order to cluster malware into families or find similarities between them. However, these works do this to try to aid forensic malware analysis, rather than trying to reduce the computational

load of dynamic IDSs. All these works were only tested on malware. My goal and implementation of my static-hybrid is fundamentally different in that it is used to distinguish malware from goodware.

[88–90] specifically aim to reduce the time dynamic analysis takes to analyze each sample. [88] uses static features to try to predict how long it will take for dynamic analysis to uncover malicious behaviors. [89] uses network analysis to specifically detect if dynamic analysis should be suspended. Both works still require the use of a sandbox, which means their tools cannot stop malware in real-time or reduce the number of programs analyzed dynamically. [90] proposed using cloud computing features to support and enhance malware analysis with the goal of reducing analysis response time, but only could improve it by 23%. Additionally, their tool is strictly a malware analysis tool, not a malware detection tool like ours.

[91] aims to maximize the value of the information obtained from dynamic analysis by using static analysis to cluster malware behavior. Their goal is to reduce duplicative runs of dynamic analysis for malware to maximize the efficiency of sandboxing. Although they do improve the efficiency of dynamic analysis, they do not aim to produce a malware detection tool such as ours.

[92] built an endpoint tool to detect maliciousness within the first five seconds of analysis. Their work is similar to ours in that the authors try to quickly detect maliciousness, but is unrelated because their tool is an endpoint tool, rather than a network IDS like ours. My work aims to prevent malware from reaching the endpoint all together. [69] similarly built an endpoint tool, but uses a set of malicious dynamic models of behavior generated during dynamic analysis to detect

malware on an endpoint. Their technology enables fast detection of malware on an endpoint, but again does not try to stop malware from reaching the endpoint all together. Endpoint tools have an entirely different set of trade offs from network-entry-point tools. In practice, the two types of tools are not in competition, with many enterprises using both types simultaneously to achieve a multi-layered defense.

[21, 22, 93, 94] all use a combination of static and dynamic analysis to either detect or forensically analyze malware. Although these works use a combination of static and dynamic analysis likes ours, they combine the analysis into one set of information with the goal of maximizing accuracy. My work uses the analysis types separately to reduce computational requirements and timeliness of dynamic analysis.

[95] proposes a two-phase classification system for detecting android malware. In the first phase, they use two separate bloom filters, one for malware and one for goodware, to classify incoming programs as malicious or benign. If their initial-phase classifiers produce conflicting results, then the android program is passed to the second, more accurate classifier. Their work differs from ours for the following reasons. First, their work is specific to android software. Second, their initial phase uses two classification tools, whereas I rely on one and use the probability of maliciousness as a method of determining which samples are passed to my dynamic component. Third, their work was only tested on 190 samples and obtained an accuracy of less than 90%. My tool was trained and tested on over 500,000 samples and obtained an overall accuracy over 99%. Further, the first phase of their tool still passed nearly 50% of the incoming programs to the second phase whereas my

tool passed less than 10%.

The following works are tangentially related to ours, but differ significantly as explained below. [96] uses static analysis to fingerprint binary code for the purpose of speeding up forensic analysis and not malware detection. [97] uses dynamic analysis to enhance static analysis for Internet malware binaries, but their technology cannot be used to reduce computational resources like ours. [98] introduces of method of smartly storing signatures on an endpoint because of the enormous amount of signatures needed to detect all malware today using a signature-based approach. [99] uses a two-phase classification tool to detect malicious obfuscation code, but performs poorly when used to detect malware.

## 4.3 My Two-phase System

### 4.3.1 Architectural Comparison VS Existing Hybrid Tools

To understand the novelty behind my work, I must first look at how existing hybrid tools operate. Figure 4.1 shows a hybrid configuration that utilizes both static and dynamic analysis to detect malware and generate behavioral reports. 100% of the the network traffic is analyzed both statically and dynamically. Although this configuration offers the best accuracy, its costs in terms of computational resources and timeliness are prohibitively high because of the use of dynamic analysis on all incoming programs.

Figure 4.2 shows a hybrid configuration that uses static analysis to filter incoming programs into two categories: benign and malicious. The traffic classified

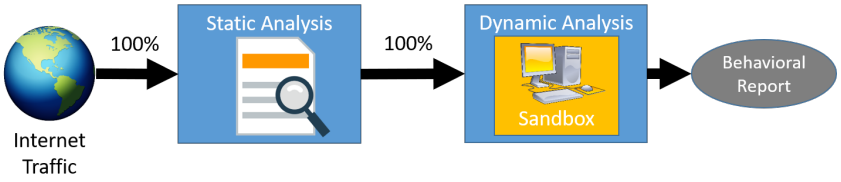


Figure 4.1: Advantages: Maximum accuracy; Disadvantages: High computational costs, decreased timeliness

as malicious generates an alert, and then is sent to the SOC to be analyzed dynamically. This model is problematic because it relies solely on static analysis to detect malware. As mentioned previously, static analysis alone can struggle to detect heavily obfuscated or packed malware, limiting its overall accuracy [6]. In addition, the alerts generated all have to be analyzed dynamically, which increases the amount of work the SOC has to perform.

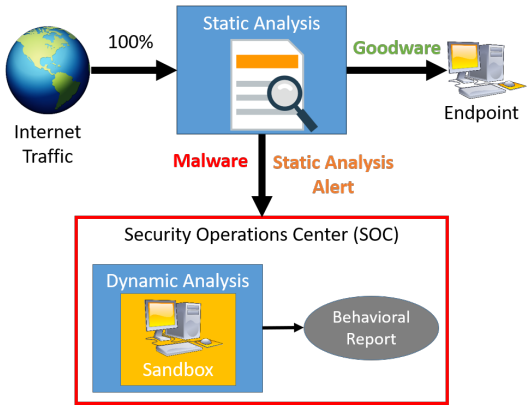


Figure 4.2: Advantages: Fast detection, some computational benefits; Disadvantages: Limited to static-only accuracy

Figure 4.3 shows my configuration and its stark improvement over existing hybrid designs. My configuration utilizes a static analysis tool to filter incoming programs into three categories: very benign, very malicious, and needs further analysis. Because of this innovative three bucket design, my tool can strategically deploy

the use of dynamic analysis only when it truly is necessary to gain an accuracy benefit.

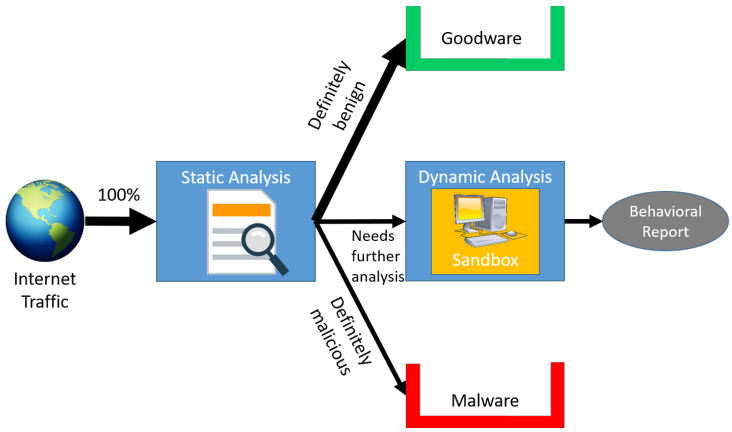


Figure 4.3: Advantages: Bandwidth and timeliness improvements, near maximum accuracy

### 4.3.2 My Design

The novelty in my work comes from my use of my static-hybrid-based machine learning tool. As shown in Figure 4.4, this static-hybrid tool divides incoming programs in real time into the following three categories based on thresholds $T_1$ and $T_2$. For the programs that the static-hybrid tool is confident is benign, the tool places them in the definitely benign bucket, bucket 1. The programs in the benign bucket bypass dynamic analysis entirely. For the programs that the tool is confident are malicious, it places them in the definitely malicious bucket, bucket 3. The programs in the malicious bucket are blocked in real-time, preventing malware from reaching the endpoint. For the programs that the tool is unsure about, it places them in the needs further analysis bucket, bucket 2. The programs in bucket 2 are passed to the dynamic analysis tool.

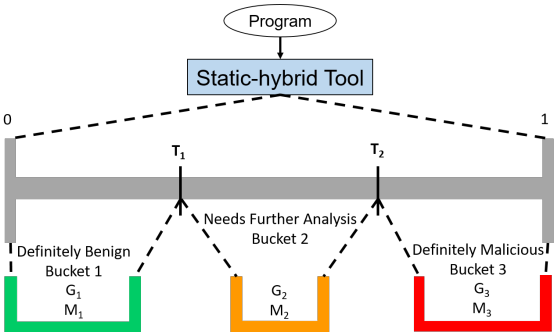

Figure 4.4: Static-hybrid Categorization

The key to my system is my static-hybrid tool. This tool has two main benefits that help my system approach the accuracy of the configuration shown in figure 4.1 with a fraction of the computational resource usage and timeliness. First, on the programs the static-hybrid tool has a confident prediction for, it is very accurate. This helps my system maintain an overall accuracy similar to hybrid systems that use dynamic analysis on all incoming programs because a large percentage of programs are correctly categorized with very low error rates. Second, the static-hybrid tool can classify a large percentage of goodware and malware into the definitely benign and definitely malicious buckets, respectively. By classifying a large percentage of goodware and malware into buckets 1 and 3 respectively, the static-hybrid tool maximizes the computational resource reduction and improvement in timeliness. Because it categorize both malware and goodware into their respective buckets, my configuration gains more computational benefits as compared to the hybrid configuration shown in figure 4.2.

For the subset of programs that the static-hybrid tool is unable to make a confident prediction for, my system's uses dynamic analysis because it can obtain behavioral information that static analysis cannot. This dynamic information can

be helpful in more accurately detecting malware because it is based on what the malware does rather than static attributes that can modified by processes like packing. In the second phase of my tool, the dynamic information obtained is combined with the static information to give my machine learning classifier the best chance at accurately detecting the remaining malware and goodware.

Static and dynamic analysis are equipped to detect malware in fundamentally different ways. This observation is key in how and why my system is built. My results show that a large portion of malware can be detected using static analysis. However, the percentage of malware that remain undetected still have massive financial and safety implications. Thus, dynamic analysis is key in modern defenses as well because it provides an additional source of information complementary to static analysis.

### 4.3.3 Computational Resources Reduction

The benign bucket is correlated to the amount of computational resources saved by not having to run dynamic analysis. As explained above, dynamic analysis is resource and computationally intensive. Especially since typically more than 99% of programs coming into a enterprise network are benign, dynamic analysis is wasteful and creates prohibitively high cost for relatively small benefits. By first learning to categorize goodware into bucket 1, my static-hybrid tool can save a tremendous amount of resources. This equates to reducing the cost of an accurate IDS for an enterprise.

### 4.3.4 Timeliness Improvement

The malicious bucket is correlated to the timeliness improvement by blocking malware in real time. In typical passive IDSs that utilize a configuration such as figure 4.1, malware is allowed through to the endpoint because dynamic analysis takes on the order of minutes to process per sample. In practice, since the percentage of programs that are indeed malicious is small, this is a compromise that is acceptable to users. However, in the cases that the program is malicious, the endpoint has to be quarantined and potentially restored from a previous backup to ensure that any infected files are not still on the machine. This is an ineffective and costly method of dealing with malware. My tool relies on static analysis, but is coupled with the power of machine learning to detect a high percentage of malware. Its ability to detect a large percentage of malware with high confidence, as shown in my results, gives it the ability to stop most malware prior to it reaching the endpoint. This saves an enterprise the time and cost of quarantining and performing damage control after a typical passive IDS would have let malware pass through.

### 4.3.5 Hard-to-Classify Programs

The needs further analysis bucket has the programs that the static-hybrid tool was unable to make a strong classification prediction for and thus are passed to the dynamic analysis tool. I refer to these programs has *hard to classify*. My belief is that the dynamic portion of my tool is able obtain information that can help it more reliably determine the correct classification for this subset of programs. My

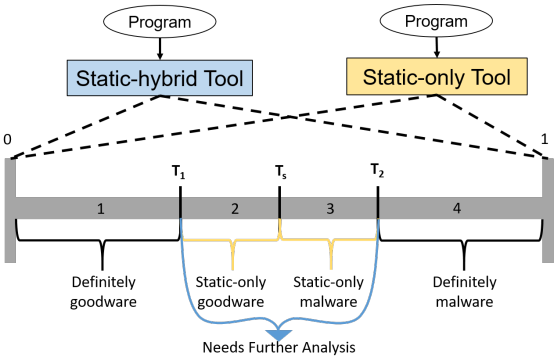


Figure 4.5: Static-only VS. Static-hybrid Comparison

conjecture is that the programs located in this bucket are harder to classify than the programs in buckets 1 and 3 and thus would benefit from dynamic analysis. The goal of my tool is to minimize the this subset of programs to maximize my system's benefits.

My results will show that on the subset of programs located in bucket 2, information from dynamic analysis is indeed valuable for more accurately classifying these programs compared to static analysis. In many cases, the increase in accuracy is dramatic, which quantifies the benefit of utilizing dynamic analysis versus static analysis. Figure 4.5 shows the overlap of my static-hybrid tool versus a tool based on static analysis alone, such as the configuration shown in figure 4.2. In a typical static-only-based tool, the programs with probabilities of maliciousness between $T_1$ and $T_s$ would be categorized as benign and the programs with probabilities between $T_s$ and $T_2$ would be categorized as malware. However, my work shows that static-only-based tools are ill equipped to reliably categorize these and thus passing them to dynamic analysis maximizes the chances of being classified correctly.

## 4.4 Implementation

### 4.4.1 Dataset

My total dataset contained 264,769 goodware and 259,356 malware Windows PE32 executables. I obtained my malware and goodware datasets through a combination of Virus Total and Virus Share [100, 101]. Both graciously allowed me to download a large number of malware and goodware. Although I cannot guarantee that my dataset is perfectly representative of all programs in the wild, I took every precaution I feasibly could while trying to amass a large, diverse set.

For my malware, I collected programs that had at least 15+ detections on Virus Total. Virus Total queries nearly 60 Anti-Virus (AV) tools to determine a program's maliciousness. I chose 15+ as an acceptable threshold to ensure the ground truth in my dataset because it meant that at least 25% of the AV tools believed this program is malicious. I ensured that my malware dataset was temporally diverse by choosing a specific number of malware per year. My malware dataset had 5,000 samples per year from the years 2001 to 2008. For the years 2009 to 2017, I chose 20,000 per year. Lastly, I chose 40,000 malware from my total dataset that had a PE timestamp that was prior to 2001 or after 2017, as these timestamps were clearly tampered with. This distribution was chosen based on distribution of a malware test set I received from a private malware analysis company. The malware chosen per year were chosen randomly.

For my goodware, I collected all of my programs from the years 2005 to 2016

and ensured that each had zero detections on Virus Total. I chose to download programs from Virus Total with zero detections because I believed that if a program was not flagged by any AV tools for more than a year, then it was most likely benign. A similar method was taken in these works [77, 92]. Additionally, this was the only way of obtaining a dataset that was large enough to effectively train and test my machine learning algorithm.

### 4.4.2 Machine Learning

My machine learning-based detection tools were trained and tested using a Multi-Layer Perceptron (MLP) neural network implemented with Tensorflow with Keras [75]. My MLP configuration, I used one hidden layer with a size equal to the input layer with dropout set to 0.5 between each. I used a learning rate of 0.01, learning rate decay of 1e-6, and Nesterov momentum set to 0.9. For my first two layers, I used a rectified linear unit activation function. I trained the MLP for 100 epochs as I observed in my training that my tool's accuracy on the validation set leveled off after 100 epochs. The final layer of my tool was a Softmax layer used to output a probability of maliciousness. I trained and tested my machine learning algorithm using a 4-fold cross validation.

The goal of my system was not to extensively test machine learning algorithms, but to show that my tool is capable of utilizing the unique strengths of both static and dynamic analysis in a previously unseen way.

### 4.4.3 Feature Set and Testing

For the static portion of my test, I generated static reports for each sample with the following class of features. My static features comprised of header data such as imported APIs, and dynamically linked libraries (DLLs), section entropy, and YARA rules. I chose to analyze APIs and DLLs based on previous work's successes using them as features. I also used YARA rules to detect signatures that were obtained from the open-source YARA-Rules project [102]. Due to the high number of unique imported APIs and DLLs found across over half a million programs, I filtered out the ones that did not occur at least 10,000 times in either the goodware or malware set. I chose 10,000 because it produced a feasible number of features that my machines could process and train on given my large dataset. This resulted in a set of 4,002 static features. The number of static features is large because each class of static features I collected, such as imported APIs, produced a large number of individual features.

For the dynamic portion of my tool, I executed each sample in a Cuckoo Sandbox for 2 minutes. Cuckoo Sandbox is an open-source sandbox manager built to dynamically analyze programs [37]. Each sandbox was run with Windows 7, 1/2 GB of ram, and 1 core. Each sandbox was given Internet access using the Whonix gateway to obscure where the HTTP requests were originating from [103].

My dynamic feature set was a superset of all the static features in addition to operating system (OS)-level and program-level behavioral signatures. The OS-level behavioral signatures were obtained using Cuckoo Monitor, Cuckoo's dynamic anal-

ysis tool that monitors the interaction between a program and the OS to build dynamic signatures. For example, Cuckoo Monitor will generate an HTTP_REQUEST signature if the program attempted to connect to the Internet. The program-level behavior was obtained from a dynamic binary instrumentation (DBI) tool based on the technologies found in [70]. The program-level features detected instruction-level obfuscations and collected instruction-level statistics. I used DynamoRio, an open-sourced DBI tool, in my implementation [73]. My dynamic feature set was 4,594 features in total.

It should be noted that my hybrid configuration, which is my main contribution, can be used by any existing static and dynamic analysis hybrid malware detection system that utilizes machine learning. Although untested, my conjecture is that my configuration's benefits are not unique to my system's feature set or implementation.

## 4.5 Results

### 4.5.1 Static VS Dynamic IDS

Here, I cover the performance of the static phase versus the dynamic phase of my two-phase system on my total dataset to prove the usefulness of dynamic analysis as a complementary addition to static analysis.

Table 4.1 below shows the following information for using a strictly static analysis-based MLP tool, and a combination of static and dynamic analysis-based MLP tool. First, it gives the detection rate, which is the percentage of malware

detected out of the malware test set. Second, it shows the false positive rate, which is the percentage of goodware that are falsely flagged as malicious. Third, it shows the area under curve (AUC), which corresponds to a machine learning classifier's receiver operating characteristic that shows the detection rate across a range of false positive rates. Lastly, it shows the F1 score, which is common machine learning metric used in practice.

| Tool | Detection Rate | False Positive Rate | F1 Score | AUC |
|---|---|---|---|---|
| Static-based MLP tool | 97.71% | 0.75% | .9850 | .9974 |
| Dynamic-based MLP tool | 99.02% | 0.75% | .9919 | .9980 |

Table 4.1: Static analysis VS. Dynamic analysis Detection Results

I used the same dataset for my dynamic tool as my static tool: 264,769 goodware and 259,356 malware. However, not all the samples executed dynamically due to issues such as dependencies not being met. As a result, I generated dynamic reports for only 195,255 goodware (70.95% of my original goodware set) and 223,352 malware (86.27% of my original malware set). When training and testing my dynamic tool, I used the same training and test set as for my static tool during the 4-fold cross validation, but only included the ones that ran. Thus, the results shown above for the dynamic-based MLP tool is based on this subset of programs that ran.

Table 4.1 shows that the false negative rate (% of missed malware) for my dynamic tool is 0.98%, compared to 2.29% for my static tool. This is a decrease

of 57.2% for the dynamic tool. These results show that dynamic analysis provides behavioral information that help a malware detection tool be more accurate than static analysis, which is why dynamic analysis is necessary in real-world tools. I observe similar increases in the F1 score and AUC compared to my static tool.

Although dynamic analysis has benefits in terms of accuracy, there are drawbacks related to the resources needed and timeliness as explained in Section 4.3. First, spawning a sandbox per sample is unscalable for many enterprise networks due to the number of programs coming in daily. Second, dynamic analysis takes on the order of minutes meaning that malware cannot be stopped in real time, and instead are allowed to pass through to the endpoint. Only after the dynamic tool determines that the program tested is malicious is it stopped and the endpoint fixed. Additionally, not all programs execute dynamically, as I experienced in my experiments.

### 4.5.2 Definitions

This subsection defines the metrics used to quantify my improvements and results of my two-phase detection tool. $G_1$ and $M_1$ are the number of goodware and malware located in bucket 1, respectively. The same notation is used for buckets 2 and 3. In my notation, $N_G$ and $N_M$ are the total number of goodware and malware respectively.

#### 4.5.2.1 Computational Requirements Reduction

I define the computational requirements (CR) of traditional dynamic IDSs as the amount of resources needed to spawn a single sandbox per sample arriving at the network. Using a typical dynamic IDS, the CR would be equal to 100%. I calculate the computational requirements of my system by Equation 4.1.

$$CR = \frac{G_2 + M_2}{N_G + N_M} \tag{4.1}$$

To calculate the CR in my system, I have to find the percentage of goodware and malware located in bucket 2 (the bucket of programs that will be passed to dynamic analysis) and divide by the total number of goodware and malware. However, I know that in the real-world, a majority of programs that arrive at a typical enterprise firewall are benign, thus the number of goodware greatly outweighs the number of malware. I can then approximate the CR of my system to be simply:

$$CR \approx \frac{G_2}{N_G} \tag{4.2}$$

#### 4.5.2.2 Timeliness Reduction

I define timeliness as the amount of time elapsed from the malware reaching the IDS to being stopped at the endpoint. I quantify it by calculating the BMR, which is how many malware are blocked and thus stopped in real time, shown in Equation 4.3. In a strictly dynamic IDS, live network traffic cannot be blocked, thus all malware that comes into the network is run on the endpoint for minutes prior

to being detected and removed. Because my static filter is fast enough for it to be used with stopping live network traffic, it can block a percentage of malware from reaching the endpoint as seen in the results below.

$$BMR = \frac{M_3}{N_M} \tag{4.3}$$

#### 4.5.2.3 Alert Generation

In typical dynamic IDSs, an alert is generated for every program flagged as malicious, which is then analyzed by a network administrator. In a dynamic scheme, the number of alerts generated are typically overwhelming and cause only a small percentage to be analyzed. In my scheme, because only a small portion of programs are dynamically analyzed, my system dramatically reduces the number of alerts sent to the network administrator.

My alert generation (AG) metric is determined by the percentage of alerts that my dynamic portion generates compared to the number of alerts a fully dynamic scheme would produce. An alert is generated only when a dynamic tool flags a program as malicious (*i.e.*, a true positive, or false positive). The formal equation is defined in Equation 4.4. First, I calculate the percentage of malware that are in bucket 2 that are detected as malicious, or my detection rate (DR) on the portion of malware in bucket 2. I then account for the percentage of goodware that are in bucket 2 that are falsely flagged, or my false positive rate (FPR) on the goodware in bucket 2.

$$AG = (DR_{M_2} * \frac{M_2}{N_M}) + (FPR_{G_2} * \frac{G_2}{N_G}) \tag{4.4}$$

### 4.5.3 Static-Hybrid Tool: Minimizing BBR

The following results are for my salient configuration, where I chose thresholds $T_1$=0.2 and $T_2$=0.999999 to minimize the BBR. I show this configuration to explain my scheme's improvements in comparison to static-only and dynamic-only analysis. Static-only analysis refers to using a strictly static IDS while dynamic-only refers to using a strictly dynamic IDS.

In Tables 4.2 and 4.3, I show the overall detection rate (DR), overall false positive rate (FPR), CR, AG, percentage of malware blocked in real-time (BMR), and percentage of benign programs wrongly blocked in real-time (BBR). The overall DR is the percentage of total malware detected as malicious; in my tool, this is by either the static and dynamic phase. The overall FPR is what percentage of total goodware were falsely detected as malicious; in my tool, this is by either the static or dynamic phase.

| Tool | DR | FPR | CR | AG |
|---|---|---|---|---|
| Static-only | 97.71% | 0.75% | 0% | 0% |
| Dynamic-only | 99.02% | 0.75% | 100% | 100% |
| Static-dynamic | **98.73%** | 0.75% | **3.63%** | **10.68%** |

Table 4.2: Static-dynamic-analysis Results for $T_1$=0.2, $T_2$=0.999999

Tables 4.2 and 4.3 shows the followings conclusions about my scheme com-

| Tool | BMR | BBR |
|---|---|---|
| Static-only | 97.71% | 0.75% |
| Dynamic-only | 0% | 0% |
| Static-dynamic | **88.98%** | **0.08%** |

Table 4.3: Static-dynamic-analysis Results for $T_1$=0.2, $T_2$=0.999999

pared to a scheme that relies solely on static analysis to detect malware. First, my hybrid scheme reduces the overall false negative rate, or the percentage of missed malware, by 44.54%. Additionally, because my tool only blocks programs in real time that are located in bucket 3, my tool only blocks 0.08% of goodware (10.7% the goodware that the static-only tool blocks). A static-only tool stops any program flagged as malicious in real time. The static-only tool does stop more malware in real time compared to my tool (97.71% versus 88.98%), but ultimately does not detect as many malware as my tool because of my use of dynamic analysis. Static-only schemes only have one chance to stop malware and if it does not, the malware is passed into the network to be executed for an indeterminate amount of time. My scheme also holds a significant advantage over static-only based detection tools because my tool can operate at almost 1/10 of the false positive rate, while ultimately detecting more malware. This greatly increases the usability and practicality of my system over existing static technologies today.

Tables 4.2 and 4.3 also shows the following conclusions about my scheme compared to a scheme that analyzes every file dynamically. The dynamic-only scheme is the most accurate, but has the costs of high computational resources, alert gen-

eration, and the inability to stop malware in real-time (as evidenced by BMR equal to 0%). My results show that my tool, for the nearly the same DR as dynamic-only, is able to use only 3.63% of the computational resources (a near 27.5X reduction), generate only 10.68% of the alerts (a 9.5X reduction), and block 88.98% of malware from entering the network (a 9X reduction). My scheme produces nearly the same overall DR as a dynamic-only scheme with a small fraction of the costs.

The reductions in computational resources and alert generation are related to increasing the scalability of dynamic analysis. By reducing computational resources by 27.5X, dynamic analysis's cost in terms of servers, virtual machines, and total analysis times is similarly reduced to ensure it can still be used even as the amount of network traffic continues to grow. Additionally, I argue that the alerts that are generated by my tool on the hard-to-classify programs are potentially more valuable than the alerts that would be generated by a dynamic-only scheme on every predicted malicious file. The alerts generated for a large percentage of malware I expect does not need human analysis because it is clearly malicious. My tool automatically produces a set of alerts for programs that were hard-to-classify and thus may need human analysis to truly understand.

My tool's BMR is an invaluable part of my system as compared to a dynamic-only scheme because a dynamic-only scheme cannot stop any malware in real-time. My scheme's ability to block 89% of malware immediately saves an immense amount of post-infection damage control and removal. This ultimately leads to reduction in cost for an enterprise to maintain a secured network.

The dynamic-only metrics shown in Tables 4.2 and 4.3 were only on the pro-

grams that executed within the sandbox (approximately 78% of programs tested). This is a detriment to using a strictly dynamic IDS because not all programs execute within a sandbox. In those cases, my static-hybrid tool proves to be invaluable in maintaining a high detection rate, while still utilizing dynamic analysis's information when available.

#### 4.5.3.1 Bucket 2 Analysis

In my scheme, my hypothesis is that current hybrid schemes do not optimally deploy dynamic analysis- meaning using dynamic information to boost confidence of prediction when static analysis alone is incapable of doing so. To prove that my system more optimally combines static and dynamic analysis, I tested the programs in bucket 2, the hard-to-classify programs, with static analysis and dynamic analysis and show the correctness of classification of goodware and malware. Correctness for malware is calculated by the DR of malware on the set of malware in bucket 2, while correctness for goodware is calculated by the FPR on the set of goodware in bucket 2. For the programs located in bucket 2 that did not able to execute dynamically, my scheme relied on static analysis for a classification.

| Tool | Bucket 2 | |
|---|---|---|
| | G (FPR) | M (DR) |
| Static | 37.85% | 6.90% |
| Dynamic | 18.83% | 96.72% |

Table 4.4: Bucket 2 Correctness Results for $T_1$=0.2, $T_2$=0.999999

Table 4.4 shows that on this subset of programs, each tool's correctness is significantly lower than their accuracies overall on goodware and malware. However, this table shows important conclusions. First, it proves that this subset of programs is hard-to-classify as evidenced by the worse accuracies. This means that the programs located in bucket 2 cannot be reliably classified by static analysis alone. Second, it shows dynamic analysis is the most capable of distinguishing these hard-to-classify programs. The dynamic analysis tool is able to nearly increase the DR of malware in bucket 2 by more than 14X while reducing the FPR by half as compared to the static-only scheme. This result shows that by utilizing my system, dynamic analysis can still be used strategically (maximizing its complementary information to boost accuracy), while incurring minimal costs as shown in the Section 4.5.3.

## 4.6 Conclusion

Malware detection is a fundamental tool necessary to prevent attacks on information and security. IDSs play a pivotal role in preventing malware attacks, but the way that current IDSs use both static and dynamic analysis are suboptimal. My two-phase system is a stark improvement over existing hybrid schemes. By understanding the unique strengths of static and dynamic analysis, my system is able to obtain an overall DR near that of a system that analyzes every file dynamically with a 27.5X reduction in computational resources, 10.5X reduction in alert generation, and a 9X reduction of malware entering a system's network. Additionally, my system can operate at a much lower FPR than a system that only relies on

static analysis, greatly increasing its practicality. Existing hybrid technologies are correct in utilizing both static and dynamic analysis when detecting malware as they are complementary to one another. However, they fail to strategically deploy these technologies to maximize efficiency and usability. My work introduces a novel method of accomplishing both.

# Chapter 5: Enhancing the Static Detection of Malware with CNNs

## 5.1 Introduction

Malware detection is at the forefront of cybersecurity research. Due to the rapid growth of malicious programs and threats, the diversity and number of malware in the world has outpaced human's ability to manually analyze and detect them. Machine learning has become intertwined with malware detection in the modern era because of its ability to decipher patterns in large amounts of data that would not be possible otherwise.

The static detection of malware initially started with static signature matching in order to detect malware. A static signature is an exact byte-for-byte match that is unique to malware that detection tools can use to confidently find malware. Recent advances with the use of machine learning have helped static analysis move away from strictly static signatures. Instead, information such as the Application Programming Interfaces (APIs) found in the program header or N-grams of opcodes has been proven to be useful for malware detection [77, 104]. Because of machine learning, large amounts of data can be analyzed for patterns indicating maliciousness within a program. Machine learning provides a probabilistic method of detecting more malware versus a static signature that typically only detects a single malware.

Amidst the recent advances in static detection of malware with machine learning, there are still areas that need improvement. The first is the often manual effort required to generate features that are useful for machine learning. Feature generation is vital in producing a highly accurate machine learning malware detection tool. This is a nontrivial task because it often times requires domain expertise and manual investigation [105]. Poor feature generation can in some ways limit machine learning's capabilities of detecting malware and goodware. Additionally, new features may have to be continually developed to deal with new malware creating a constant need to study in-depth the latest malware trends. The second issue relates to the prominence of using features such as N-grams or other strictly presence-absence features that can cause overfitting resulting in less generalizable results [2]. Malware detection has largely moved away from traditional hash-based signatures, but using features such as N-gram combinations of Windows APIs can be seen as a similar variant to hash-based signatures. They are only more effective because of the computing power of machine learning. More advances have to be made in order to reduce the time and knowledge necessary to create quality features and help improve a malware detection tool's ability to generalize and detect new malware.

To address the issues outlined above, I propose the use of static program disassembly and Convolutional Neural Networks (CNNs) in order to automatically generate machine learning features that are effective at differentiating between goodware and malware. I apply Natural Language Processing (NLP) methods to the static disassembly of both malware and goodware with the hypothesis that the opcode se-

quences found in programs can be treated as sentences would be in natural language processing. With the use of CNNs, I show that my tool is able to automatically generate features that embed raw opcode sequences without any manual intervention, and improve the accuracy of a tool based on existing static analysis features.

My work has four main advantages. First, the use of CNNs to automatically generate useful features reduces the time and effort required compared to manually-generated features. As the number and diversity of malware continue to grow, the need for more scalable feature generation is evident. My method is completely automatic, which presents a scalable and useful method of future malware detection. My tool requires no human expertise in identifying which opcode sequences are malicious or benign. In addition, because the feature vectors based on opcode sequences are generated by the model itself, the features are optimized to improve the performance of the detection tool. Second, I show that when my generated features are added to an existing static analysis tool's feature set, the percentage of missed malware decreases by about 40%. This is a significant finding because it shows that the CNN is capable of reducing the work necessary to produce high-quality features. Additionally, my tool does not prefilter the dataset in any way, meaning that both packed and obfuscated malware are included in my set. Previous methods that relied on the reliable uncovering a malware's API sequences or true opcodes can be defeated by packing or obfuscation techniques. My tool shows the ability to perform well despite such malware. Third, my work shows in a simulated zero-day malware test that the addition of my features to a typical static analysis-based set of features decrease the number of missed malware by more than 50%.

This is significant because other simple exact-pattern matching features have shown to be prone to overfitting and thus perform poorly on unseen malware [2]. Lastly, analysis into the results show that malware that have the same generated feature from the CNN model are indeed similar. This is a somewhat expected, but exciting finding because it shows that there is promise in using CNNs for automatically identifying similar malware and their malicious code.

My approach works as follows. I first obtain the static disassembly of a Windows PE32's executable sections. I then extract only the opcodes in program order because it is my belief that the opcodes are more important to the behavior of the malware versus the operands, which may change arbitrarily. I then feed in the opcode sequences into my CNN, which produces a feature vector representation of the input sequence. The feature vector representation of the input opcode sequence encodes the relationship between opcodes found in the sequence. This feature vector representation is then fed into a back-end machine learning algorithm for classification between goodware and malware.

The rest of the paper is structured as follows. Section 5.2 gives an overview of PE32 executable as well as the methodology behind my tool. Section 5.3 details the implementation of my work, while Section 5.4 covers the results. Section 5.5 covers the related work.

## 5.2 Static Program Structure Feature Set

### 5.2.1 PE32 Program Structure

A Windows PE32 executable is structured in the following way. First, there is a header, which contains information such as where the code and data sections reside along with imported Application Programming Interfaces (APIs) and Dynamically Linked Libraries (DLLs). The remaining portions of the executable are its individual sections, which can be code or some form of data.

Code sections are typically marked as read/execute in the header, while data sections are typically read-only or read/write. In x86, the code sections sometimes can contain data as well, which makes the process of disassembly difficult. In addition, because x86 is a complex instruction computing set architecture with variable length instructions, there may be multiple legitimate instruction sequences within the same set of bytes. These two facts make static disassembly of x86 code hard to do reliably. Especially in the case of malware, static disassembly can be tricked or fooled by packing or obfuscation.

As my proposed method below relies on static disassembly, the problems with it mentioned above are concerning. However, my research and results show the following. I found in practice that although malware can be heavily packed and obfuscated in order to trick static disassemblers, I was still able to obtain code traces from over 93% of malware tested. This could be improved by choosing a more robust static disassembly tool. Also, because the extracted code is simply a

basis for the CNN's generated features, my tool's performance supports the claim that it can find effective feature vectors that are able to differentiate malware and goodware despite the presence obfuscated or packed programs.

### 5.2.2 Base Static Features

The first set of static features I obtain are what I call my *base* features. This set of features contains header information and Cuckoo static information. My static header-related features included imported APIs and DLLs, section entropy, and language. Features such as the presence or absence of specific APIs or language were boolean features whereas features such as section entropy were value features. Cuckoo Sandbox is a commonly used open-source sandbox tool used to analyze malware [37]. It also contains a static module for detecting static attributes of the program under study. I chose to include those as features as well to make my base feature set as robust as possible. These were standard static features I found in previous work.

As shown in my related work section, a majority of the static analysis-based malware detection papers relied on PE header information and some variant of the presence or absence of Windows APIs and DLLs similar to my base static features. Although machine learning has greatly enhanced the ability to detect all types of malware accurately and at scale, these types of features are not a significant improvement in comparison to the traditional hash-based signatures because these types of features can still require domain expertise (to filter out which APIs/DLLs

are relevant) and some manual analysis to be used effectively to detect malware. Machine learning has automated some of the process, but my work shows that further automation can take place to build accurate detection tools without any manual analysis or expertise.

### 5.2.3 Exact or Near-exact Pattern Matching Features

The second portion to my static feature set are exact or near-exact pattern matching rules. I chose to use the Python library PEID and YARA rules in practice as examples of these rules [102]. Both of these tools use exact or near-exact byte-level matches in order to detect specific attributes of programs such as which type of packer or compiler was used. The byte-level matches can occur in code, the header, or the data section. Although my opcode sequence CNN model also looks for similarities in the code, these methods are still distinct for reasons explained in subsection 5.2.4. These tools are targeted at identifying specific attributes of malware that can be useful to an analyst. Because many of these rules are specific to malware, they have been shown to be effective as boolean features for differentiating malware and goodware [70]. Thus, I included them in my feature set to increase the performance of my static analysis detection tool.

In order to properly validate the value of my automatically generated features from opcode sequences, I felt that it was necessary to include this set of features to prove two things. First, I want to show that my generated features still add value in terms of accuracy despite some overlap in methodology between the opcode sequence

CNN model and strict pattern matching. In order to test this, I had to include a set of strict rule-based features to show improvement. Second, I wanted to further demonstrate that quality features can be generated despite the lack of knowledge about which opcode sequences were malicious versus benign. By including the PEID and YARA rules as features that were developed by experts with specific knowledge, the improvement in accuracy in my results better reinforces this idea.

As the number of malware and goodware continue to grow, hard-coded rules to detect attributes of malware will become harder to maintain and develop. Even now, out of the nearly 2,000 YARA and PEID signatures I found in my malware set, more than 80% had a hit rate of less than 0.1%. As malware detection and analysis continue to try to scale with the rapid growth of malware, a more scalable, automatic, and efficient method of feature generation that does not require the expertise of the ever-changing attributes of malware is paramount.

### 5.2.4 Opcode-Based Feature Generation and Training

A natural way to view a program is as a sequence of instructions. In this section, I discuss how machine learning, specifically CNNs, can be used to extract features from the sequence of instructions that make up an executable program.

The process of feature extraction begins with static disassembly of the executable. Disassembly is the process of turning machine language code into human-readable assembly language. Since this process occurs statically, the returned sequences of assembly language opcodes are in program order. Our process only

extracts the opcodes out of the disassembly because operands, such as registers, can be changed arbitrarily and are less important to the core operation of the program. By extracting only the opcodes, the storage space necessary and computing time also decreases because of the smaller amount of raw data extracted.

Once each program is represented as a program order sequence of opcodes, any sequence-based ML model, such as CNNs using one-dimensional convolution, can be trained to recognize sequences of opcodes that occur frequently in either malware or goodware. This allows these sequence-based models to discern between the two classes of programs using only an input sequence of opcodes.

CNNs have classically been applied to sequential data mainly in Natural Language Processing (NLP) for tasks like sentence classification and sentiment analysis [106]. From a high-level this application of CNNs to malware detection can be viewed as a similar task to sentence classification where the input tokens are a sequence of opcodes rather than words and the output classes are malicious and benign rather than sentence topics.

CNNs, like most ML models, are designed as a composition of non-linear functions, called layers, that allow for each layer to create a representation of the input based on the output of the previous layer. In the case of CNNs, an *embedding layer* initially assigns each opcode a random real-valued vector. Following the initial embedding layer is a series of convolutional layers. These convolutional layers generate filters, short sequences of vectors, and convolve these filters with the output of the previous layer. Each layer ends by applying a non-linear element-wise *activation function* to the output of the convolution. Intuitively, this procedure can be seen

as a fuzzy matching operation between the layer's filters and the layer's input. If a filter's sequence of vectors are similar to a subsequence in the layer's input, then the convolution operation creates a larger output than if the sequences are dissimilar. Succinctly, each filter can be described by the following input-output rule:

$$y_i = f(W \cdot X_{i:i+h} + b) \tag{5.1}$$

where $y_i$ is the $i$th output of the convolution, $W$ is the filter, $X_{i:i+h}$ is the sequence of embeddings for opcodes $i$ through $i+h$, and $f(\cdot)$ is an activation function. After applying a series of convolutional layers, CNN classifiers typically create a set of output class probabilities by first performing a pooling operation across the final convolutional layer's filter outputs and then applying fully-connected layers as seen in regular feedforward neural networks.

The training procedure for CNNs uses the backpropagation algorithm with any variant of stochastic gradient descent. This algorithm runs a batch of samples from the training dataset through the model, producing output classifications. These outputs are compared to the known class labels associated with each sample and a loss function, usually cross entropy for classification tasks, measuring the model's deviation from the ground truth labels is evaluated. The model parameters are then updated by taking the gradient of the loss function with respect to all of the parameters so that each parameter can be adjusted in the direction that locally minimizes the loss function.

Superficially, this idea of using a CNN for opcode-level analysis is reminiscent of N-gram opcode analysis or other code-based pattern matching. However, as seen

by previous research, N-gram analysis at the byte level tends to overfit to training data and CNNs operating on the byte-level were shown to be more robust to minor changes to the input bytes seen in polymorphic malware [2]. Since N-gram analysis at the byte level is also sensitive to single byte changes, it is reasonable to hypothesize that similar results would be seen at the opcode level.

Additionally, the high-level interpretation of what a CNN does matches well with human intuition of how programs work. Unlike other sequence-based ML models such as Recurrent Neural Networks, a CNN layer only considers a small, spatially relevant number of input tokens when generating an output value. This is seen in equation 5.1 where the $i$th output of a filter is a function of input tokens $X_{i:i+h}$. This matches with the intuition that only short sequences of contiguous opcodes are actually relevant to one another, especially when dealing with a program-order disassembly.

## 5.3 Implementation Details

### 5.3.1 Dataset

My total dataset contained 101,847 goodware and 96,753 malware Windows PE32 executables. I obtained my malware and goodware datasets through a combination of Virus Total and Virus Share [100, 101]. Both graciously allowed me to download a large number of malware and goodware. Although I cannot guarantee that my dataset is perfectly representative of all programs in the wild, I took every precaution I feasibly could while trying to amass a large, diverse set. This includes

not prefiltering my dataset in any other way than ensuring I had a temporally diverse dataset. Analysis on my malware and goodware set showed that 52% and 25% were packed, respectively.

For my malware, I collected programs that had at least 15+ detections on Virus Total. Virus Total queries nearly 60 Anti-Virus (AV) tools to determine a program's maliciousness. I chose 15+ as an acceptable threshold to ensure the ground truth in my dataset because it meant that at least 25% of the AV tools believed this program is malicious. I ensured that my malware dataset was temporally diverse by choosing a specific number of malware per year. My malware dataset had 2,000 samples per year from the years 2001 to 2008. For the years 2009 to 2017, I chose 8,000 per year. Lastly, I chose 16,000 malware from my total dataset that had a PE timestamp that was prior to 2001 or after 2017, as these timestamps were clearly tampered with. This distribution was chosen based on distribution of a malware test set I received from a private malware analysis company. The malware samples for each year were chosen randomly.

For my goodware, I collected all of my programs from the years 2005 to 2016 and ensured that each had zero detections on Virus Total. I chose to download programs from Virus Total with zero detections because I believed that if a program was not flagged by any AV tools for more than a year, then it was most likely benign. A similar method was taken in these works [77, 92]. Additionally, this was the only way of obtaining a dataset that was large enough to effectively train and test my machine learning algorithm. I chose the goodware dataset based on a similar distribution as the malware.

### 5.3.2 Disassembly

The first step in my process is producing the static disassembly of each program in my dataset. I chose to use the Linux utility ObjDump to produce the static disassembly of the executable sections found in each program. ObjDump uses linear sweep to disassemble a program. Linear sweep disassembles the code regions sequentially assuming there is no interleaved data. Although linear sweep can be prone to errors, I chose to use ObjDump because of its ability to obtain sizable sequences of opcodes. In practice I saw that ObjDump was successful in uncovering opcode sequences more than 93% of the time. By choosing a more robust tool, this number may be improved upon.

My results and work shows that despite the pitfalls of linear sweep disassembly, my model is able to produce opcode sequence feature vectors that are still capable of differentiating malware from goodware. Additionally, packed or obfuscated malware that produce erroneous disassembly may be somewhat beneficial because it could produce uncommon opcode sequences that my tool is able to attribute to malware.

### 5.3.3 Machine Learning

I implemented my entire tool with Python3 and Tensorflow with Keras. I used a basic ensemble configuration to test and compare my different feature sets. My ensemble is two layers, a set of base classifiers that feed into a second-level classifier that takes the output probabilities of the base classifiers as input. I chose to use an ensemble configuration due to its success in previous works. Each base classifier is

described below.

The first base classifier is trained on a set of static features I call my *base static* features. This set of features is described in 5.2.2. This set of features totaled 1,302 features in total. I chose to train an Multi-Layer Perceptron (MLP) neural network to train on the base static features. I found that through testing the MLP was able to handle the large number of features well in a reasonable amount of time. For my MLP configuration, I used one hidden layer with a size equal to the input layer with dropout set to 0.5 between each. I used a learning rate of 0.01, learning rate decay of 1e-6, and Nesterov momentum set to 0.9. For my first two layers, I used a rectified linear unit activation function. I trained the MLP for 45 epochs as I observed in my training that my tool's accuracy on the validation set leveled off after 45 epochs. The final layer of my tool was a Softmax layer used to output a probability of maliciousness. I trained and tested the MLP using a 4-fold cross validation.

The second base classifier is trained on the set features described in 5.2.3, which I call the *rule-based* features. These features were all boolean features and generated from the YARA open-source project as well as the Python library PEID [102]. My dataset had 2,002 collective features from these tools. I used the same MLP configuration as above to train and test the accuracy of this feature set. This model was also trained for 45 epochs as the validation accuracy leveled off. A 4-fold cross validation was used here as well.

The third base classifier was my opcode sequence CNN model described in 5.2.4. I chose to feed in sequences of length 10,000 opcodes after testing various

sequence lengths because it performed the best by a slight margin with reasonable training times. During the training phase, I found that in practice training on a randomly selected contiguous chunk of 10K opcode sequences from within a program in a given class was more efficient versus training on the every 10K contiguous chunk of each program. Due to the large number of samples in my dataset, there were a subset of programs with extremely long disassemblies, which made training times highly impractical. Thus, the CNN model was trained with contiguous chunks of 10K opcode sequences from within the malware or goodware training samples. I randomly chose 10K opcode sequences within each binary versus choosing the first 10K opcode sequence to prevent bias and overfitting on the code located at the entry point. During the testing phase, I feed each 10K opcode sequence chunk from the entire testing sample into the CNN and backend MLP to get a set of prediction probabilities. To get a final determination for a given sample, I average the output probabilities. I originally tried taking the max, but found that it caused a high number of false positives. In practice, I found that this configuration of training and testing performed best while maintaining practical training times.

For configuration of the CNN itself, it uses a set of two gated convolutional layers followed by the max pooling layer along the opcode dimension. Each gated convolutional layer uses two sets of convolutional filters. The output of the first set of filters is passed through a sigmoid activation function that maps the output values to the interval ¡0,1¿. These values are then element-wise multiplied by the outputs of the second set of convolutional filters. This gated convolution configuration follows the approach in [107]. The intermediate output from the max pooling layer is an

automatically generated feature vector with a 1x100 dimension. This generated feature vector was then fed into the same MLP configuration as the other base models mentioned above to get a final prediction. For programs that did not have at least 10K opcodes, I padded the opcode sequence with a null opcode reserved specifically to pad my sequence vectors. I used a 4-fold cross validation here.

The 2nd-layer classifier I chose to use to train on the output probabilities of the base classifiers was the Random Forest algorithm. I found that through experimentation that Random Forest was the best performing 2nd-level classifier for my testing scenario. I built the Random Forest classifier with 300 estimators from the SkLearn Python library.

## 5.4 Results

### 5.4.1 The Addition of Opcode Sequence-based CNN Features

The first test results I collected shown in table 5.1 shows the accuracy of two ensemble configurations. The second row shows the first (labeled base+pattern) configuration where the base classifiers include the MLP trained on the base static features and the MLP trained on the pattern-matching features. The second configuration (labeled base+pattern+opcode) is when the base models include the two previously mentioned models plus my opcode sequence CNN model. The second column abbreviated DR stands for detection rate. The third column has the F1 score, which is a common machine learning accuracy metric. The fourth column has the AUC, which is short for Area Under the Curve, of the Receiver Operating

Characteristic (ROC) curve. The last column has the False Negative (FN) reduction or the reduction in the number of malware missed.

| Base Models | DR | F1 score | AUC | FN Reduction |
|---|---|---|---|---|
| Base+Pattern | 90.91% | 0.959 | 0.988 | – |
| Base+Pattern+Opcode | 94.56% | 0.971 | 0.994 | 40.2% |

Table 5.1: Ensemble Results with Various Base Classifiers at a 1% FPR

The table of results showcase the following conclusions. First, it shows the relatively good performance of using the base static features and the pattern-based feature set. The AUC is nearly 99%, which is inline with we saw in other prior works on similar sized datasets using static analysis. The more significant finding is the 40% reduction in the false negative rate or the number of missed malware when adding the automatically generated opcode sequence features into the first ensemble configuration. The false negative reduction is calculated by subtracting the false negative rate of the second model (5.44%) from the false negative rate of the first model (9.09%) and then dividing the resulting number (9.09% - 5.44% = 3.65%) by the false negative rate of the first model (3.65% / 9.09% = 40.2%). The AUC and F1 score similarly have gains of 50% and 29.3%. The value added to the existing static detection tool is evident.

Looking closer at the results, my hypothesis that adding this set of automatically generated opcode sequence-based features adds value in the context of malware detection is confirmed. Furthermore, after the analysis of my malware dataset and goodware dataset revealed that 52.4% and 24.92%, respectively, were packed, the

fact that the CNN model was still capable of producing quality features based on disassembly produced by linear sweep that helped differentiate more malware and goodware than using other standard static features alone is significant. This shows the promise of using CNNs to automatically produce features that are valuable for bettering malware detection while reducing the manual labor involved in generating features specific to malware.

Figure 5.1 shows the ROC curve for both ensemble configurations. As shown, the second model with the CNN-generated features outperforms the other configuration across a multitude of false positive rates.

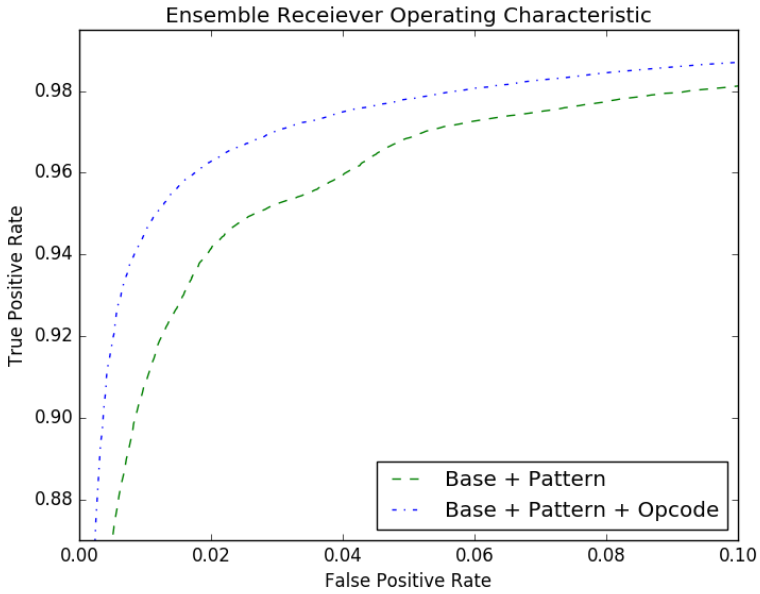


Figure 5.1: ROC plot for Ensemble with $0\% \leq \text{FPR} \leq 10\%$

### 5.4.2 Zero-day Malware Detection

In order to further test the value of my opcode sequence-based CNN features, I ran a simulated zero-day malware detection test. I trained my base models only on

goodware and malware from before 2014 within my dataset and tested on goodware and malware that were compiled after 2015. This results in 73,475 malware and 83,988 goodware in the training set and 23,946 malware and 17,993 goodware in the test set. The results are shown in table 5.2.

| Base Models | DR | F1 score | AUC | FN Reduction |
|---|---|---|---|---|
| Base+Pattern | 72.25% | 0.923 | 0.966 | – |
| Base+Pattern+Opcode | 87.04% | 0.951 | 0.986 | 53.3% |

Table 5.2: Simulated Zero-day Test: Ensemble Results with Various Base Classifiers at a 1.5% FPR

The purpose of this test was to confirm prior work's claims that features such as exact pattern matching or other boolean features such as the presence or absence of specific APIs were prone to overfitting while showing that CNN-based features are less prone. Overfitting would cause superficially high accuracies on datasets that contained similar samples in the test set and training set (as seen in most malware datasets) and significantly lower accuracies on samples unseen before. As shown in the second row, the detection rate, F1 score, and AUC of the model without the opcode sequence-based features drop off dramatically. The detection rate on this test shows a decrease of 22.1% when compared to its detection rate of 92.75% with a 1.5% FPR on the standard 4-fold CV test. Although the model with the opcode sequence-based features also has a dip in performance, it is only a loss of 9.0% compared to its detection rate of 96.71% with a 1.5% FPR on the standard 4-fold CV test. In other related work studied, I saw a similar dip in performance

when testing across time because malware is constantly changing to adapt to recent updates, platforms, vulnerabilities, and detection techniques.

When compared head to head, the model with the additional opcode sequence-based features reduces the false negative rate of the model without it by over 50%. This helps confirm that the opcode sequence features generated by the CNN are useful in reducing the inability of other feature sets, which are solely based on boolean features or exact matches, to generalize better.

This time test overall shows that there is still significant work to be done to be able to build malware detection tools that can reliably detect zero-day malware. As the landscape of malware continues to change rapidly, the information malware detection tools draw upon has to change and be updated equally rapidly. The only way to do this is through automation. Although the opcode sequence-based feature vectors are not the complete answer, the promise they showed in improving the accuracy without needing prior knowledge on what the latest malware trends were is significant and interesting area of future research.

### 5.4.3 CNN Robustness Test

To further confirm that the CNN is capable of robustly handling minor changes in the raw input sequence and performing some form of fuzzy matching, I conducted an experiment on a randomly chosen test set of 24,213 malware. I changed a varying percentage of the opcodes in the input sequence and measured the change in the output probability of maliciousness. I randomly chose the opcodes to alter within

each input sequence using a random number generator for the location and ensured that there was no repeats in locations chosen. I also used a random number generator to come up with the opcode that replaced the existing opcode. I varied the percentage of opcodes randomly replaced in the following increments: 5%, 10%, 20%, 30%, 40%, 50%. The results in the change of the output probability are shown in Figure 5.2.

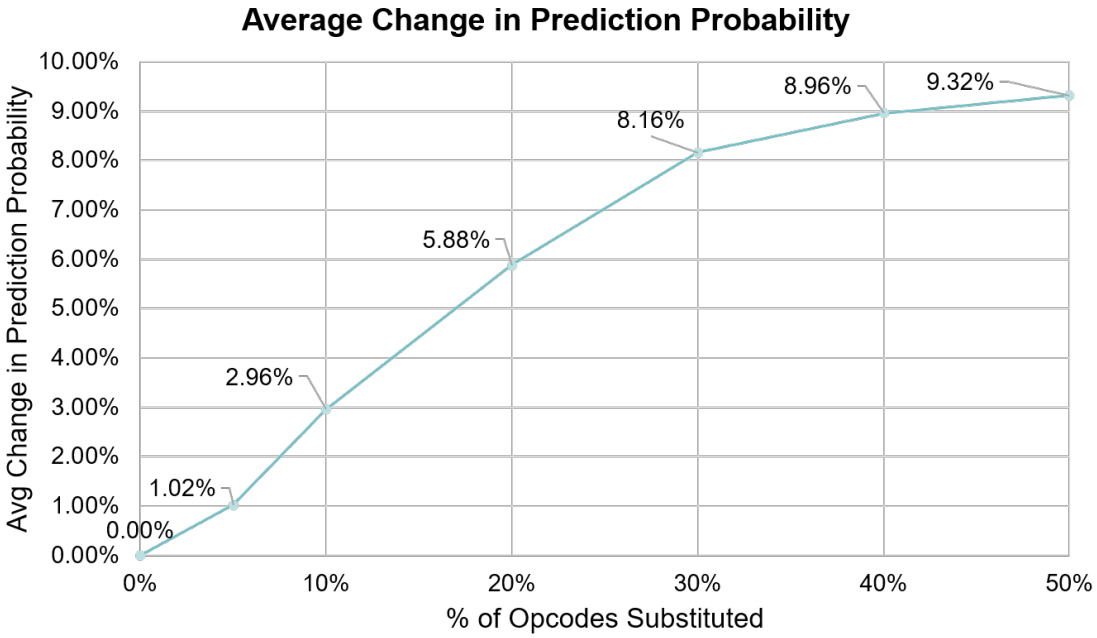


Figure 5.2: Robustness Test Results

The results confirm that CNNs are indeed more than capable at handling small to fairly large changes in the input sequence without the changes having a large effect on the output probability. Despite up to 50% of the input opcodes in the input sequence being changed, the CNN model's probability of maliciousness only decreased by an average of 9.32% compared to its original probability of maliciousness on the original unchanged input sequence. This is a desirable quality in

malware detection because malware can have arbitrary changes in its code sections that may not directly affect the overall function of the malware. In these scenarios, it is beneficial to still classify these similar, but not identical opcode sequences as sequences originating in malware. This robustness gives our CNN model a significant advantage over exact-pattern matching feature sets. Exact-pattern matching feature sets are more strict and thus could be more heavily affected by small arbitrary changes in the malware, resulting in a higher likelihood of overfitting and poor generalization results.

### 5.4.4 Similarity Analysis and Future Work

At an intuitive level, the CNN model is performing a fuzzy match of opcode sequences found in goodware and malware automatically. Using this information and by analyzing the resulting generated feature vectors, a simple similarity test can be performed. By storing all of the generated feature vectors by the CNN model for malware and goodware, I ran a test to see how similar two different programs were if their respective input opcode sequences mapped to the same feature vector. I chose to focus on the generated feature based on the first 10K opcode sequences of each program as I found that it was the most reliable in delivering the most quality matches.

I first found out the most commonly generated features that were element-wise equal and near equal. The most commonly generated feature was found 1,023 times, all occurring from malware. Taking a closer look, I found that all 1,023

of these malware had the same list of 41 imported APIs, giving high confidence that these malware had very similar if not the same opcode sequences. This is somewhat of an expected finding as intuitively the CNN was finding similar opcode sequences that were specific to goodware or malware. I further investigated the next ten most common embeddings and their associated occurrences from goodware or malware. Each set of programs that had the same generated feature were all exclusively malware and imported the same set of Windows APIs respectively. In the small case studies done, the confirmation that the CNN model is finding similar programs based solely on opcode sequences without the use of any prior human expertise is significant because it opens up the prospect of potentially using this system not only as a malware detection engine, but as a malware analysis tool. The biggest benefit here is that this was done automatically.

I am not proposing this work as a general malware classification tool, as there are many other works that focus exclusively on that area. However, I do think there is potential for this framework to be used in the future to automatically find malware that have very similar code. Although this tool is not as general as other classification tools, based on my limited analysis, there is potential for future research to look more heavily into this technique.

## 5.5 Related Work

[4] used Echo State Networks (ESNs) and Recurrent Neural Networks (RNNs) in order to automatically generate features for the purpose of malware detection.

However, their schemes relied on the dynamic sequence of specific Windows APIs called whereas my tool only requires the static disassembly of the program, which is faster and easier to obtain. Additionally, the authors had to first condense the low-level APIs into distinct, high-level events to consolidate behaviors such as opening a file since there is more than one API to do so. My approach does not require the same level effort in terms of preprocessing or prior knowledge.

[3] used a CNN to produce features from a raw sequence of bytes from a program. Their approach consisted of feeding in the whole binary (up to 2 million bytes), disregarding the inherent structure of the binary, into the CNN in order to generate a feature and ultimately provide a classification between goodware and malware. Their work on a comparably sized dataset achieved less than 90% overall accuracy whereas my CNN model alone is able to achieve 93% overall. Because their work treats each byte as a sequence unit, their work could be less resilient to arbitrary changes made such as changes in the data section. The data section of a program, especially if it is packed or encrypted in some way will be significantly different from compilation to compilation, which can cause issues in their scheme. My model takes into account the inherent structure of the binary and recognizes that the opcode sequence is a better representation of a program, which is backed up by my results.

[108] introduces an automatic string signature generation system in order to detect specific strings in malware with near-zero detections in goodware. Although their goal of malware detection is similar to ours, their programs need to be unpacked prior to being fed into their system. This is a significant disadvantage because of the

time and effort required to accurately unpack programs. Their paper reveals that less than 55% of their original malware dataset were successfully unpacked. Also, their system uses exact matches, which has a low false positive rate, but can cause overfitting.

Many works such as [109, 110] have used various types of neural networks in order to generate features for the purpose of malware classification, not malware detection. [109] uses CNNs and RNNs with the dynamic system call trace as input to identify which family of malware a sample belonged to. [110] uses an unsupervised deep belief network that takes as input dynamic behavior from malware and outputs a malware family classification. My work is different from this class of work because my automatic feature generation is for the purpose of malware detection.

Other works such as [45, 49, 50, 71, 77, 104] propose machine learning-based malware detection tool based on static features such as PE imports, strings, and other metadata. As previously mentioned, these types of features are more prone to overfitting than perhaps a CNN model would be and thus when run in a time split experiment, their performance drops off rapidly. This is also supported by our findings in our results.

Work such as [105, 111] use similar natural language processing methods for malware detection goals, but differ from my work because [111] focuses on capturing malicious payloads and [105] focuses solely on android malware.

## 5.6 Conclusion

The need for further advancements in the static detection of malware with machine learning is necessary to handle the ever-growing cyber threats. The use of CNNs and raw opcode sequences is innovative because it reduces the need for malware and malicious code expertise and manual analysis. Instead, it relies on the CNN's ability to generate quality features that are helpful to existing static detection tools. My results show that with the added CNN model's automatically generated features, it can reduce the false negative rate of a tool trained on existing features by over 40% and by over 50% on a simulated zero-day test. Additionally, our CNN model is capable of robustly handling minor changes in the input sequences of opcodes, resulting in a fuzzy matching of opcode sequences from malware or goodware. This is an advantage over previous work that relied on exact or near-exact matches and proves to be more robust in practice. Although the features generated from the CNN model are not the complete solution to malware detection, it shows promise is finding opcode sequences specific to goodware or malware while maintaining the flexibility needed to handle modern malware.

# Chapter 6: Conclusion

Malware is a decidedly hard problem to solve. Windows PE32 executables are only a small slice of the total problem that malware presents. My Ph.D. research has shown that both properly understanding malware and the ways in which it is often detected leads to useful research. By targeting fundamental behaviors, such as the ways that malware attempts to hide from static detection tools, my work shows that malware can be identified by their attempts using dynamic program-level analysis. With the understanding of how malware detection tools are developed and the time it takes to find malicious indicators, my CNN work is able to reduce the time and knowledge necessary to produce high-quality features indicative of malware. Lastly, by properly understanding how current malware detection tools are setup, my work combining static and dynamic analysis methods shows that the tradeoff is not linear and improves upon existing hybrid detection schemes.

# Bibliography


[1] Symantec Corporation. Internet security threat report. 21:5, apr 2016.

[2] Richard Zak, Edward Raff, and Charles Nicholas. What can n-grams learn for malware detection? In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 109–118. IEEE, 2017.

[3] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[4] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[5] Philip O'Kane, Sakir Sezer, and Keiran McLaughlin. Obfuscation: The hidden malware. *Security & Privacy, IEEE*, 9(5):41–47, 2011.

[6] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[7] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Still: Exploit code detection via static taint and initialization analyses. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 289–298. IEEE, 2008.

[8] Kevin A Roundy and Barton P Miller. Binary-code obfuscations in prevalent packer tools. *ACM Journal Name*, 1:21, 2012.

[9] Daniel A Quist and Lorie M Liebrock. Visualizing compiled executables for malware analysis. In *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pages 27–32. IEEE, 2009.

[10] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[11] Danny Quist and V Smith. Covert debugging circumventing software armoring techniques. *Black hat briefings USA*, 2007.

[12] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[13] Matias Madou, Ludo Van Put, and Koen De Bosschere. Understanding obfuscated code. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 268–274. IEEE, 2006.

[14] Jesse C Rabek, Roger I Khazan, Scott M Lewandowski, and Robert K Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 76–82. ACM, 2003.

[15] Joe Stewart. Ollybone: Semi-automatic unpacking on ia-32. In *Proceedings of the 14th DEF CON Hacking Conference*, 2006.

[16] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109. IEEE, 2009.

[17] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[19] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, (2):32–39, 2007.

[20] Rob Pike, Bart Locanthi, and John Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Softw., Pract. Exper.*, 15(2):131–151, 1985.

[21] Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. 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, 2013.

[22] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300. IEEE, 2006.

[23] Piotr Bania. Generic unpacking of self-modifying, aggressive, packed binary programs. *arXiv preprint arXiv:0905.4581*, 2009.

[24] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *Information Hiding*, pages 232–248. Springer, 2006.

[25] Saumya Debray and Jay Patel. Reverse engineering self-modifying code: Unpacker extraction. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 131–140. IEEE, 2010.

[26] Jonas Maebe and Koen De Bosschere. Instrumenting self-modifying code. *arXiv preprint cs/0309029*, 2003.

[27] Sudeep Ghosh, Jason Hiser, and Jack W Davidson. Software protection for dynamically-generated code. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 1. ACM, 2013.

[28] Microsoft Coporation. What is data execution prevention? may 2016.

[29] Oleh Yuschuk. Ollydbg, 2007.

[30] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441. IEEE, 2007.

[31] Lingyun Ying, Purui Su, Dengguo Feng, Xianggen Wang, Yi Yang, and Yu Liu. Reconbin: Reconstructing binary file from execution for software analysis. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 222–229. IEEE, 2009.

[32] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53. ACM, 2007.

[33] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[34] Chandra Prakash. Design of x86 emulator for generic unpacking. In *Assocation of Anti-Virus Asia Researchers International Conference. Seoul, South Korea*, 2007.

[35] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *Usenix Security*, 2007.

[36] Aditya Kapoor. *An approach towards disassembly of malicious binary executables*. PhD thesis, University of Louisiana at Lafayette, 2004.

[37] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. The cuckoo sandbox, 2012.

[38] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[39] Thomas Hungenberg and Matthias Eckert. Inetsim: Internet services simulation suite, 2013.

[40] National Institute of Standards and Technology. National software reference library. Online.

[41] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129. ACM, 2015.

[42] Yi-Min Wang, Binh Vo, Roussi Roussev, Chad Verbowski, and Aaron Johnson. Strider ghostbuster: Why itsa bad idea for stealth software to hide files. Technical report, Technical Report MSR-TR-2004-71, Microsoft Research, 2004.

[43] Tony Rodrigues. Extracting known bad hash set from nsrl, feb 2010.

[44] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[45] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[46] P Vinod, Vijay Laxmi, and Manoj Singh Gaur. Scattered feature space for malware analysis. *Advances in Computing and Communications*, pages 562–571, 2011.

[47] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *Journal of Machine Learning Research*, 13(Apr):949–979, 2012.

[48] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[49] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. 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, 2001.

[50] Igor Santos, Javier Nieves, and Pablo Garcia Bringas. Semi-supervised learning for unknown malware detection. In *DCAI*, pages 415–422. Springer, 2011.

[51] Eitan Menahem, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Improving malware detection by applying multi-inducer ensemble. *Computational Statistics & Data Analysis*, 53(4):1483–1494, 2009.

[52] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[53] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 301–310. IEEE, 2008.

[54] Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. *Journal in computer virology*, 5(4):283–293, 2009.

[55] Igor Santos, Yoseba K Penya, Jaime Devesa, and Pablo Garcia Bringas. N-grams-based file signatures for malware detection. *ICEIS (2)*, 9:317–320, 2009.

[56] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[57] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE, 2004.

[58] Robert Moskovitch, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. Unknown malcode detection via text categorization and the imbalance problem. In *Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on*, pages 156–161. IEEE, 2008.

[59] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. Unknown malcode detection using opcode representation. *Intelligence and Security Informatics*, pages 204–215, 2008.

[60] Robert Moskovitch, Clint Feher, and Yuval Elovici. Unknown malcode detectiona chronological evaluation. In *Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on*, pages 267–268. IEEE, 2008.

[61] Olivier Henchiri and Nathalie Japkowicz. A feature selection and evaluation scheme for computer virus detection. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 891–895. IEEE, 2006.

[62] Boyun Zhang, Jianping Yin, Jingbo Hao, Dingxing Zhang, and Shulin Wang. Malicious codes detection based on ensemble learning. *Autonomic and trusted computing*, pages 468–477, 2007.

[63] Yuval Elovici, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. Applying machine learning techniques for detection of malicious code in network traffic. In *Annual Conference on Artificial Intelligence*, pages 44–50. Springer, 2007.

[64] Yi-Bin Lu, Shu-Chang Din, Chao-Fu Zheng, and Bai-Jian Gao. Using multi-feature and classifier ensembles to improve malware detection. *Journal of CCIT*, 39(2):57–72, 2010.

[65] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[66] Swapna Vemparala. Malware detection using dynamic analysis. 2015.

[67] MG Kang, P Poosankam, and H Yin Renovo. A hidden code extractor for packed executables [c]. In *Workshop on Recurring Malcode (WORM 2007)*, 2007.

[68] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 45–60. IEEE, 2010.

[69] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[70] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled ElWazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic

obfuscation in malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 97–118. Springer, 2017.

[71] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent pe-malware detection system based on association mining. *Journal in computer virology*, 4(4):323–334, 2008.

[72] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[73] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[74] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[75] Francois Chollet. Keras: deep learning library for theano and tensorflow. 2015.

[76] Tomasz Kojm. Clamav, 2004.

[77] Joshua Saxe and Konstantin Berlin. 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, 2015.

[78] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 35–44. ACM, 2015.

[79] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1871–1878. ACM, 2010.

[80] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2012.

[81] Georg Wicherski. pehash: A novel approach to fast malware clustering. *LEET*, 9:8, 2009.

[82] Engin Kirda and C Kruegel. *Large-Scale Dynamic Malware Analysis*. PhD thesis, PhD Dissertation, Technical University of Vienna, 2009.

[83] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[84] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[85] Silvio Cesare and Yang Xiang. A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 721–728. IEEE, 2010.

[86] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *International Symposium on Engineering Secure Software and Systems*, pages 35–43. Springer, 2010.

[87] Vinod P Nair, Harshit Jain, Yashwant K Golecha, Manoj Singh Gaur, and Vijay Laxmi. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, pages 263–269. ACM, 2010.

[88] Sean Kilgallon, Leonardo De La Rosa, and John Cavazos. Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. In *Resilience Week (RWS), 2017*, pages 30–36. IEEE, 2017.

[89] Toshiki Shibahara, Takeshi Yagi, Mitsuaki Akiyama, Daiki Chiba, and Takeshi Yada. Efficient dynamic malware analysis based on network behavior using deep learning. In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–7. IEEE, 2016.

[90] Osamah L Barakat, Shaiful J Hashim, Raja Syamsul Azmir B Raja Abdullah, Abdul Rahman Ramli, Fazirulhisyam Hashim, Khairulmizam Samsudin, and Mahmud Ab Rahman. Malware analysis performance enhancement using cloud computing. *Journal of Computer Virology and Hacking Techniques*, 10(1):1–10, 2014.

[91] Matthias Neugschwandtner, Paolo Milani Comparetti, Gregoire Jacob, and Christopher Kruegel. Forecast: skimming off the malware cream. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 11–20. ACM, 2011.

[92] Matilda Rhode, Pete Burnap, and Kevin Jones. Early stage malware prediction using recurrent neural networks. *arXiv preprint arXiv:1708.03513*, 2017.

[93] James B Fraley. *Improved Detection for Advanced Polymorphic Malware*. PhD thesis, Nova Southeastern University, 2017.

[94] PV Shijo and A Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[95] Priyadarshani M Kate and Sunita V Dhavale. Two phase static analysis technique for android malware detection. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*, pages 650–655. ACM, 2015.

[96] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 341–355. Springer, 2017.

[97] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[98] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G Andersen. Splitscreen: Enabling efficient, distributed malware detection. *Journal of Communications and Networks*, 13(2):187–200, 2011.

[99] D Swathigavaishnave and R Sarala. Detection of malicious code-injection attack using two phase analysis technique. *Pondicherry Engineering College Puducherry, India*, 2012.

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

[101] J-Michael Roberts. Virus share.(2011). *URL https://virusshare. com*, 2011.

[102] Yara-rules. `https://github.com/Yara-Rules/rules`, 2017.

[103]

[104] Zane Markel and Michael Bilzor. Building a machine learning classifier for malware detection. In *Anti-malware Testing Research (WATeR), 2014 Second Workshop on*, pages 1–4. IEEE, 2014.

[105] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778. ACM, 2016.

[106] Cicero dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.

[107] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

[108] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *International workshop on recent advances in intrusion detection*, pages 101–120. Springer, 2009.

[109] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[110] Omid E David and Nathan S Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE, 2015.

[111] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, pages 171–184, 2008.